Noname manuscript No.
(will be inserted by the editor)

Risk of Generating Technical Debt Interest: A Case Study

Georgios Digkas · Apostolos Ampatzoglou · Alexander Chatzigeorgiou · Paris Avgeriou · Oliviu Matei · Robert Heb

Received: date / Accepted: date

Abstract In the context of Technical Debt (TD), interest refers to the additional maintenance effort and associated costs that stem from the very existence of TD items in a system. The generation of TD interest may make the system break, in the sense that too much interest, generated from a design hotspot, can constitute the system unsustainable. In this paper, we consider the generation of interest as a risk and present a metric (namely Interest Generation Risk Importance–IGRI) to quantify this risk. Subsequently, we validate this metric in two ways. First, we explore whether the metric can be used to effectively prioritize TD items. Second we investigate if adding new code reduces the risk of interest generation. Based on our findings, we have observed that: (a) IGRI is capable of efficiently prioritizing TD items; and (b) that the new code that is introduced in the system is usually less risky for producing interest, compared to legacy code.

Keywords technical debt · maintainability · new code · clean code

1 Introduction

In software engineering, the concept of Technical Debt draws an analogy between shortcuts in development and taking out a loan [14]. In particular, the metaphor considers that a software development organization (intentionally or unintentionally) limits the development time/resources through shortcuts, and thus saves

Work reported in this paper has received funding from the European Union H2020 research and innovation programme under grant agreement No. 780572 (project: SDK4ED).

G. Digkas, P. Avgeriou
Institute of Mathematics and Computer Science, University of Groningen, Netherlands
E-mail: g.digkas@rug.nl, paris@cs.rug.nl

G. Digkas, A. Ampatzoglou, A. Chatzigeorgiou
Department of Applied Informatics, University of Macedonia, Greece
E-mail: g.digkas@uom.edu.gr, a.ampatzoglou@uom.edu.gr, achat@uom.edu.gr

O. Matei, R. Heb
Holisun SRL, Baia Mare, Romania
E-mail: oliviu.matei@holisun.com, robert.heb@holisun.com

a specific amount of money (amount of loan–***TD Principal***) [1,2]. This benefit comes with an associated cost, as the product is released with sub-optimal quality, leading to the occurrence of excessive maintenance costs [19]; such costs are termed ***TD Interest*** and include the additional costs for bug fixing, understanding the existing code, adding new features etc. [1,2]. While TD Principal is deterministic, TD interest is probabilistic: we are not sure how frequently and to what extent a software artifact will change in the upcoming versions (thus generating interest). The probability of an artifact to generate interest is termed ***TD Interest Probability*** [32]. The generation of interest plays a crucial role for the impact of TD on software maintenance. Modules that are rarely maintained, do not cause real problems along software evolution, even if they suffer from high TD; paying back the TD in such cases might be unnecessary. On the contrary, modules with high levels of TD that are often maintained, can cause severe overhead when performing future changes. Such overheads can be so large that in some cases can constitute the maintenance of the software system unsustainable. Based on this, we consider the generation of interest as a *risk* that threatens software maintenance.

In this study we propose a metric, namely **Interest Generation Risk Importance (IGRI)**, to estimate the risk of interest generation. According to Barry Boehm [9], the importance of a risk can be calculated as the product of its impact and likelihood to occur. In the case of IGRI, the likelihood of the risk corresponds to interest probability, whereas the impact to the amount of technical debt interest that it will generate. The proposed metric can be useful in a number of ways; in this study we validate two of them. The first is to assist TD Prioritization, i.e. the priority to refactor a software artifact [24]. Artifacts that pose a more important risk to generate TD interest, would be more urgent for refactoring, so as to prevent excessive maintenance costs. The second is to assess the effect of writing clean new code on the technical debt evolution of the system. If new code is less risky to generate interest, the sustainability of the system can be improved by the addition of clean new code. The clean code paradigm is supported in the literature as an alternative to refactoring for the improvement of software quality [26], and it tends to be preferable from the developers' side, as a means of containing technical debt [5].

The research outcomes reported in this study has been conducted in the context of the SDK4ED[1] project, funded by the European Union's Horizon 2020 research and innovation programme. The goal of the project is to investigate tradeoffs between optimizations applied to improve Technical Debt, Security and Energy dissipation in software intensive systems. Furthermore, the SDK4ED platform [20] aims at assisting decision making with respect to investments on software improvements. The assessment of artifacts which pose a high risk of generating TD interest outlined in this study, is aligned with the overall goal of the project to narrow down the recommended refactoring opportunities. Choosing among optimizations to mitigate software vulnerabilities detected through static analysis [35], to improve performance [34,33,18] and energy consumption [41,25,43,25], to improve software maintainability [11,38,22,27,16], and to manage architectural technical debt [31] is a non-trivial task. Research has proved the existence of interrelations between these qualities [28,36,37] rendering the extraction of the best possible sequence of software refactoring subject to a Multi-Criteria Decision Mak-

[1] `https://sdk4ed.eu/`

ing (MCDM) analysis which has been implemented in the SDK4ED platform, as well as an approach for treating refactorings as a financial investment [3].

We organize the rest of the paper as follows. In Section 2, we present: (a) related work on technical debt prioritization; and (b) background work on software risk management. The framework that we use for calculating Technical Debt Interest and Interest Probability, as well as the proposed metric for assessing the importance of interest generation as a risk are introduced in Section 3. In Section 4, we present the empirical design through which we explore the two aforementioned usage scenarios of the IGRI metric. In Section 5 we answer the research questions, which we discuss in Section 6, by interpreting the results and providing implications for researchers and practitioners. Finally, in Section 7 we present the threats to validity of this study accompanied by mitigation actions, whereas in Section 8 we conclude the paper.

2 Related Work and Background Information

This section discusses related work and background material necessary for facilitating the readability of this study. In particular, we discuss relevant studies: i.e., studies on technical debt prioritization; and we discuss background concepts from the software risk management literature.

TD Prioritization is the technical debt management activity that aims at ranking the TD items that have been identified, based on their urgency to be resolved [24]. According to Li et al. [24], TD Prioritization has been studied in 18% of the TD research corpus. TD Prioritization methods can be discussed under two perspectives: based on the concepts used as inputs, as well as, based on the approach used for prioritization per se. With respect to inputs [32], TD prioritization can be performed based on the main TD metrics (i.e., TD principal, interest, or interest probability). With respect to approach, existing methods for TD prioritization can be categorized into four main classes: (a) cost-benefit-based approaches; (b) principal-based approaches; (c) portfolio management approaches; and (d) interest-based approaches.

Software Risk Management is the process that aims at: (a) identifying the risks in software development; (b) determining their importance; and (c) suggesting mitigation actions [7]. According to Boehm, there are three main categories of risks: project risks, product risks, and business risks [10]. Among these categories, the generation of Technical Debt Interest falls in the product risk category: i.e., "*risks that affect the quality or performance of the software being developed*". In this paper, we focus on ***Risk Analysis*** (see Sommerville [40]): risk analysis aims at assessing the likelihood and consequences of all risks identified in a system. Therefore, the rest of this sub-section focuses on how risks can be assessed. In the literature, there are two main schools for risk assessment: categorical risk assessment and continuous risk assessment. Regarding categorical risk assessment, according to Sommerville [40], risk analysis relies on judgement and experience to find the probability of a risk (rare, unlikely, possible, likely, or almost certain) and the effects of the risk (catastrophic, major, moderate, minor, or negligible). Based on this the project managers generate a table according to seriousness of risk and update it during each iteration of the risk process, as shown in Figure 1.

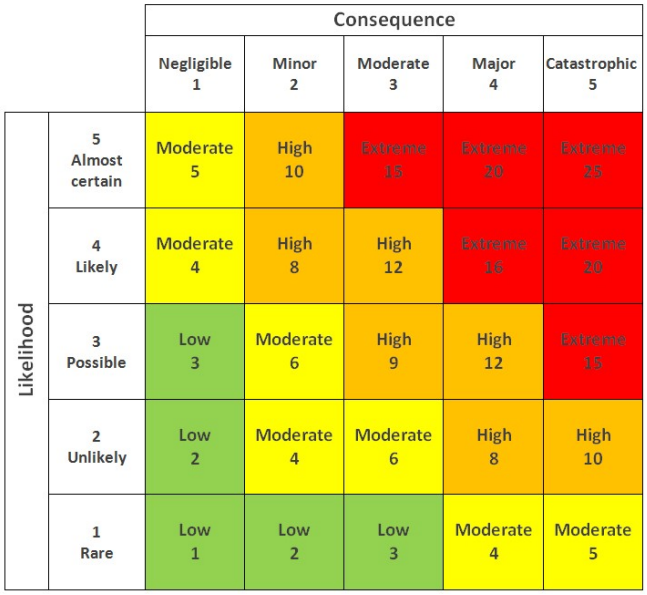

Fig. 1: Risk Assessment Matrix

With respect to continuous risk assessment, in a seminal work of software project management, Boehm [10] introduced the basic principles of software risk management. As part of risk analysis, he suggests that risk exposure (also termed as risk impact) can be calculated as the product of the probability of unsatisfactory outcome and the loss caused by the unsatisfactory outcome.

3 Assessing the Risk of Generating Interest

This section focuses on tailoring the software risk management concepts to fit the technical debt metaphor. IGRI is meant to quantify the impact of a possible change in a specific artifact that suffers from technical debt, by assessing the probability of this artifact to change and the amount of interest that is going to be generated, upon such a tentative change. The rationale of the metric relies on the risk importance formula, as proposed by Boehm [8]. The rest of this section describes the way that TD Interest and TD Interest Probability are calculated.

$$IGRI = Interest \times InterestProbability \tag{1}$$

3.1 Technical Debt Interest

The estimation of TD Interest is a challenging endeavor as interest refers to the "additional" maintenance costs, incurred by inefficiencies in software; whereas the nominal software maintenance effort (i.e. the effort that would have been required

in case the system had zero technical debt) is unknown. The calculation of Technical Debt Interest in this study relies on the FITTED framework, which has been proposed [2] and empirically validated in our previous work [4, 42]. FITTED attempts to calculate interest by estimating the level of "sub-optimality" of any given software artifact. In particular, FITTED assumes that the effort required to maintain a sub-optimal artifact is analogous to the ratio of the maintainability of the two artifacts:

$$Effort_{opt} = Effort_{act} \times \frac{maintainability_{act}}{maintainability_{opt}} \tag{2}$$

Based on its definition, Technical Debt Interest can be obtained by subtracting the optimal from the actual effort resulting in the following formula:

$$TDInterest = Effort_{act} \times (1 - \frac{maintainability_{act}}{maintainability_{opt}}) \tag{3}$$

Maintainability. To calculate the maintainability of a software artifact, we first identify a set of other artifacts [4]) that can be considered (structurally) similar. Artifact similarity is assessed based on their lines of code, number of methods, cognitive complexity. Second, we calculate the optimal value of selected metrics among the set of similar artifacts. The best metric scores (highest or lowest values, based on the type of the metric) are assumed to characterize the hypothetical *'optimal'* which the artifact under study could potentially reach. A simplified example is outlined in Fig. 2.

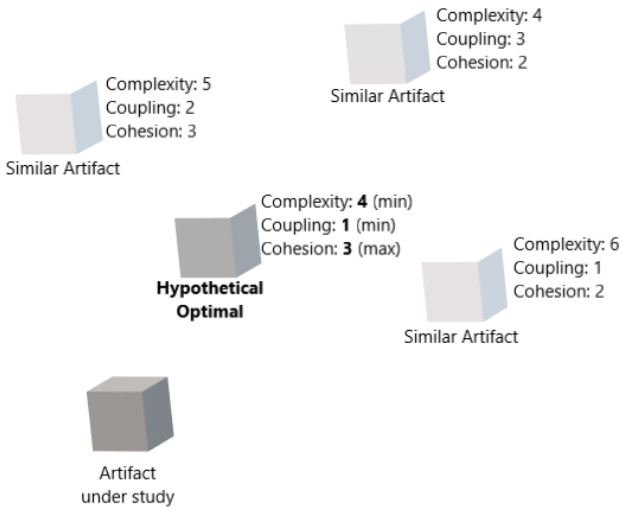

Fig. 2: Notion of hypothetical optimal among similar artifacts

Finally, we assess maintainability as the average ratio of each metric score of the artifact under study, against the optimal value. As maintainability metrics, we

have used ten well-established maintainability predictors (Depth of Inheritance Tree–DIT, Number of Childen Classes–NOCC, Message Passing Coupling–MPC, Response for a Class–RFC, Data Abstraction Coupling–DAC, Lack of Cohesion of Methods–LCOM, Cyclomatic Complexity–CC, Weighted Method per Class–WMPC, Lines of Code–LOC, and Number of Properties–SIZE2), from the metric suites of Chidamber and Kemerer [12], and Li and Henry [23]. The aforementioned list of metrics has been acknowledged by Riaz et al. [29] as the most prominent maintainability predictors.

Maintenance Effort. Predicting how a system will evolve is a non-trivial task, in the sense that none can foresee with certainty the changes that will be requested by customers, bug fixing activities, etc. Hence, we follow a simple, yet relatively reasonable approach and assess future maintenance effort, based on historical change data. In particular, for each artifact we calculate the average number of code lines which have been introduced, removed or changed, for every transition from one revision to the next, and use this as a proxy of maintenance effort in each future commit. Finally, we project this average maintenance effort to future releases of the analyzed artifact. A similar approach was employed in various other research works [13, 21].

3.2 Technical Debt Interest Probability

Interest probability is calculated based on past maintenance data. To this end we use the Percentage of Commits in which a Class has Changed (PCCC) metric [6]. Despite the fact that the Effort of Maintenance and PCCC are related, the two measures correspond to different views of the same phenomenon, in the sense that Maintenance Effort captures lines of code (i.e. the average extent of change), whereas PCCC a number of commits (frequency of changes). Thus, we consider them independent and suitable for being used in the same calculation.

4 Case Study Design

The case study is designed and reported based on the linear-analytic structure as described by Runeson et al. [30]. This section presents the study design in detail.

4.1 Research Goals and Questions

The aim of this study is two-fold: (a) to assess whether the proposed metric can perform effective prioritization of TD items; and (b) to examine the risk of interest generation posed by new code. Based on the above, we have set two research questions:

RQ1: Is IGRI able to prioritize TD items similarly to an expert? To answer this research question, we calculate IGRI for all classes of a project and we record the urgency to fix TD (specifically to improve the quality of individual classes), based on the expert opinion of software engineers. A correlation analysis between the IGRI values and the expert opinions could validate or invalidate IGRI as a suitable prioritization indicator. In case IGRI is able to resemble the

expert opinion with a strong correlation, we would be able to resolve an important scalability problem in TDM, since experts cannot afford to assess hundreds or even thousands of artifacts manually. Using IGRI, they would instead get automated suggestions on which TD items to check first for refactoring opportunities.

RQ2: Does new code pose a lower risk (in terms of IGRI) for generating TD interest? This question is refined through two sub-questions to distinguish between the quantity and quality of new code: ***(RQ2.1)*** *Is IGRI of a component related to the amount of new code introduced to that component over time?* and ***(RQ2.2)*** *Is IGRI of a component related to the average quality of new code introduced to that component over time?*

This research question focuses on new code introduced over time, which as explained in Section 1, can be a promising technical debt reduction approach, if the new code is of high quality. To answer this research question we need to first separate new from modified code in each commit, and then capture the extent as well as the quality of the new code. As a second step, we need to perform the FITTED analysis, and calculate IGRI. Finally, a correlation analysis will be performed to answer this research question. The outcome of the analysis can inform researchers and practitioners whether the introduction of (clean) new code can lead to a more sustainable evolution, that generates less technical debt interest. We conjecture that the more and the cleaner the new code that is added in a component, the less the risk for that component to produce interest. Subsequently, one could advocate the writing of clean new code as a way to decrease the risk of generating interest, effectively reversing the negative effect of TD.

4.2 Cases and Units of Analysis

This study is an embedded multiple case study, in which the case is an existing software system (written in Java), and the units of analysis are its classes. The system that we have analyzed is MaQuali that is developed by Holisun SRL. MaQuali is a quality management system (ISO 9001) supporting the handling of business processes. It consists of approx. 100 classes (more than 150,000 lines of code) and has been maintained for more than a decade.

4.3 Data Collection

To address the research questions the following process has been followed. In the **first step**, we initially analyzed the MaQuali source code with the SDK4ED toolkit[2] and **quantified IGRI (Interest Generation Risk Importance)** for every class of the software. Then, we aggregated the results at the level of packages. Next, we randomly picked[3] 10 packages and asked Holisun's engineers to provide a ranking of these packages in terms of maintenance risk. This process has led us to the dataset outlined in Table 1. The first column corresponds to the name

[2] `https://sdk4ed.eu`

[3] The selection process was as follows. First, we sorted the packages by IGRI, then we have demarcated 10 areas (bins), each one containing 10% of the packages. Finally, we selected 10 software packages, randomly picked from each one of the 10% bins.

of the package, the second column to interest (per commit), the third to interest probability, and the fourth to IGRI.

Table 1: TD Interest Assessment of MaQuali Packages

Package	Interest	Interest Probability	IGRI
fr.icms.db	57.67 $	0.50	28.83
fr.icms.sorters	0.12 $	0.00	0.00
fr.icms.models	16.22 $	0.25	4.05
fr.icms streams	0.17 $	0.12	0.02
fr.icms.mail	16.50 $	0.50	8.25
fr.icms.renderers	0.46 $	0.25	0.11
fr.icms.printing	1.71 $	0.12	0.21
fr.icms.graph	0.70 $	0.25	0.17
fr.icms.ui	9.97 $	0.25	2.49
fr.icms.os	4.55 $	0.25	1.13

As a **second step**, we **asked the software engineers** of Holisun that focus on MaQuali maintenance to rank the aforementioned packages, based on the following question: "*Please rank the aforementioned packages (ties are acceptable—however, not preferable) in terms of the risk that their maintenance might lead to extreme delays. As maintenance, please consider the time that you spend for adding a new requirement, for fixing a bug, etc. In this question, consider not only the time required for one maintenance action, but also how frequently you need to maintain them. Assign 1 to the package that is the least risky and 20 to the most risky packages*". Packages have been shuffled for each respondent, while the assessments of each package, based on the SDK4ED platform was hidden from the engineers. The analysis of the respondents' answers (5 software engineers) have been aggregated.

As a **third step**, we have performed the **analysis of new code technical debt**, similarly to our earlier work [15]. For a software system evolving through a number of revisions we track new methods introduced either in entirely new packages or in existing classes. We then compute the quality of these new methods in terms of their technical debt by linking identified technical debt issues to these methods. Note that we use the concept of $\mathbf{TD}_{density}$, which is the technical debt of these methods normalized over their size in lines of code. $\mathrm{TD}_{density}$ enables the comparison of technical debt between artifacts of different sizes (such as new methods vs. the already existing system).

The process for analyzing git repositories (such as the repository of MaQuali) is briefly outlined next:

***Step 1**: Commit Retrieval*

1. For the target project we obtain all revisions from the main branch.
2. The revisions are sorted chronologically. In case of commits with more than one parent, we have extracted the nodes leading to the longest path between the commit node under examination and the start node (i.e. the only node with no parent). This choice avoids any (chronological) inconsistencies among revisions and at the same time, the longest path yields the largest number of commits to be analyzed yielding a higher granularity for the analysis.
3. To avoid unnecessary computations, the analysis ignores transitions between successive commits without modifications to source code files.

***Step 2**: Linking technical debt issues with methods*

Each technical debt issue is linked to the method in which the issue resides according to the following sub steps:

1. All issues per revision are retrieved.
2. We link the retrieved issues to the method in which they reside based on the line of code reported for each issue.

***Step 3**: Tracking new methods*

We identify the introduction of new methods in the following way:

1. *New Methods in New Files:* For files which are new in each revision, we tag all of their methods as new. The methods of each file are obtained with the use of their Abstract Syntax Tree.
2. *New Methods in Modified Files:* For files which have been modified in a revision, we identify entirely new methods using the Gumtree Spoon AST Diff tool[4].

***Step 4**: Obtaining the impact of new methods to the system's $TD_{density}$*

As the goal is to isolate the impact of new code (i.e. new methods) on $\text{TD}_{density}$ per transition from one commit (at time point *t-1*) to the next (at time point *t*) we employ the following formula:

Impact of new methods

$$\Delta TD_{density}(new) = \frac{TD_{t-1} + TD_{new(t)}}{LOC_{t-1} + LOC_{new(t)}} - TD_{density}(t-1) \tag{4}$$

Based on the aforementioned process, the following dataset has been developed: each row represented a class, whereas the columns held the following information:

[V1] Package Name
[V2] IGRI
[V3] Expert opinion of Holisun Software Engineers on the Risk of the Class
[V4] Average LoC added as new code in the history of the package
[V5] Average impact of new code on the $\text{TD}_{density}$ of the package

4.4 Analysis

The aforementioned data have been analyzed using descriptive statistics and by Spearman Correlation in pairs. To answer RQ1 we use the pair [V2]-[V3], for RQ2.1 the pair [V2]-[V4], whereas for RQ2.2 the pair [V2]-[V5]. Especially for RQ2.2, we have transformed [V5] variable to a categorical one (positive or negative impact) and provided additional analysis.

5 Results

In this section, we present the results of the case study, organized by research question. In particular, in Section 5.1, we present the results on the ability of IGRI to predict the risk of software packages to produce high interest. Subsequently, in Section 5.2, we use the newly proposed index to assess the impact of new code on the risk of producing technical debt interest.

[4] https://github.com/SpoonLabs/gumtree-spoon-ast-diff

5.1 Ability of IGRI to Prioritize TD Artifacts

As a means for validating the ability of IGRI to estimate of the risk of a package to generate Technical Debt Interest, we contrast the metric to the perception of stakeholders on the risk of package maintenance to lead to extreme risks. To achieve this goal, since: (a) the two variables have a different value range; and (b) we focus on prioritization instead of prediction, we preferred to treat the two variables as ordinal ones. Thus, we transformed them to the rank that corresponds to a specific value (i.e., the highest IGRI is assigned the value 1; whereas the lowest IGRI is assigned the value 10). To visualize the ability of IGRI to consistently rank packages, based on their risk to produce interest (**metric property**: consistency [39]), in Figure 3, we present a scatter plot.

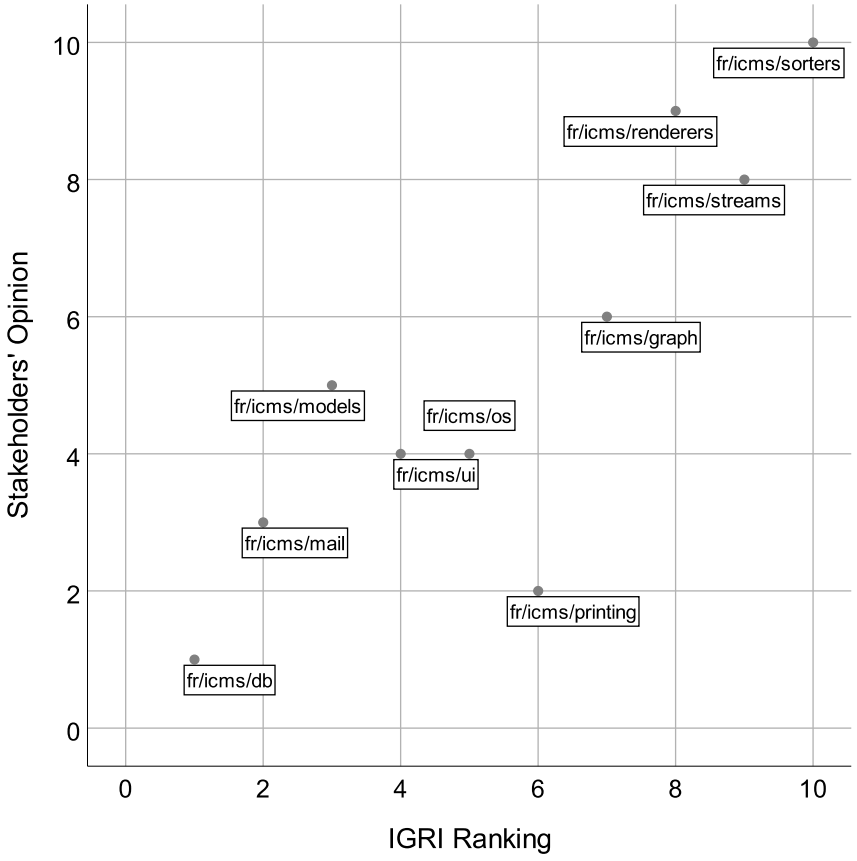

Fig. 3: Risk Ranking Consistency of IGRI and Perception of Stakeholders

As it can be observed from Figure 3, the ranking is almost consistent (the dots are close to the main diagonal line), with only some exceptions. The majority of deviations is by one rank, with only one exception (**package**: `fr.icms.printing`).

The specific package is ranked as *high risk* according to stakeholders, but as *low risk*, based on IGRI. According to a lead developer of the MaQuali software: "*the printing package was difficult to maintain and keep sustainable, because of the lack of strong printing support for java programs (especially custom HTML printing)*". The fact that the interest probability of this package is quite low, we can infer that the opinion of stakeholders on risk is more related to Technical Debt Interest (i.e., maintenance difficulty) rather than maintenance frequency.

Regarding the extreme cases (highest or lowest IGRI), the packages `fr.icms.db` and `fr.icms.sorters` are correctly characterized as high and low risk by both stakeholders and IGRI (the dot on lower left is ranked with 1 from both IGRI and practitioners, as well as the dot on the upper right is ranked with 10 from both IGRI and practitioners). Quoting a practitioner: "*On the one hand, the `fr.icms.sorters` package is rarely maintained because the code is pretty basic (no complex logic inside) and classes inside are used as basic components in lots of other parts of the application, so most of maintenance was made on the beginning of development phase of the project. On the other hand, the difficulty in maintaining the `fr.icms.db` package comes from the fact that new requested features implied the modification of the underlying database structure*". This confirms that both ease of maintenance and maintenance load are deemed as important by the practitioners.

To explore if the aforementioned results are statistically significant, we performed a Spearman Rank correlation analysis. The results (**correlation coefficient**: 0.827 and **sig**: 0.003) of the test suggest that the two ranks are very strongly correlated (and statistically significant). Thus, an IGRI-based prioritization can safely subsume [39] the ranking that an experienced practitioner would provide to packages in terms of risk to generate Technical Debt Interest. This outcome is significant, in the sense that IGRI calculation is automated; therefore, it can easily scale at large codebases, and is unbiased from stakeholders experience. Thus, inexperienced developers can use the ranking and identify maintainability challenges similarly to more experienced developers.

5.2 Relation of IGRI and new code

In this section, we use the IGRI metric, so as to explore the relation between new code and the risk of generating TD Interest. New code has been discussed in the literature as an alternative to refactoring, for reducing the amount of technical debt [17, 44]. To this end, we explore: (a) if the average percentage of new code that is accumulated in a package along evolution, is associated with a decrease or increase of IGRI–RQ2.1; and (b) if there is a relation between IGRI and the quality of the new code–RQ2.2.

Regarding RQ2.1, we first explore if the average percentage of new code (in all versions) against all lines of code of the package, is correlated to IGRI of the package (calculated as the average value of the IGRI score of its classes). The results suggest that there exist a very strong (**correlation coefficient**: -0.745 and **sig**: 0.012) negative relation, that is statistically significant. The negative sign of the relation suggests that the less new code is introduced in a package, the higher the risk of the corresponding package generating interest. To visualize the aforementioned relationship, in Figure 4, we present the boxplots of IGRI

for each percentile (quartile) of new code density. For instance the first percentile corresponds to packages that 0% - 25% of their code in each version (on average) is new. Based on Figure 4, the median IGRI for the packages of this group is approx. 8 (**mean value**: 13.84); whereas the median IGRI for packages in which new code accounts for 26% - 50% of their codebase the median is almost zero (**mean value**: 1.09).

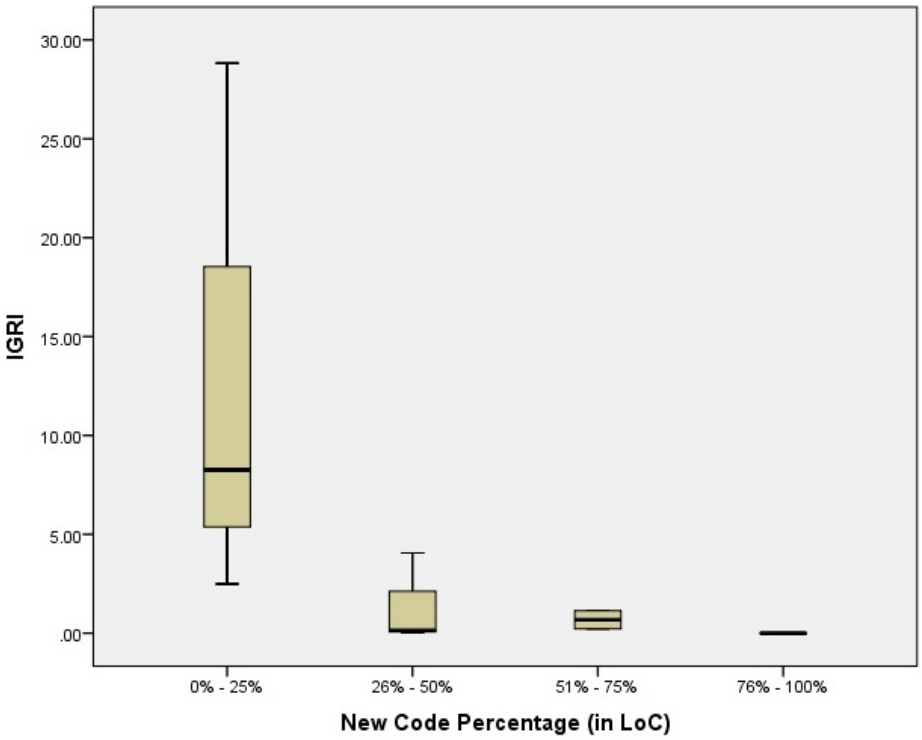

Fig. 4: Percentage of New Code and Risk of Producing Interest

Regarding RQ2.2, rather than focusing on the amount of new code that is added, we focus on the quality of the new code. Thus, we correlated the $TD_{density}$ of the new code with IGRI. The results of the Spearman correlation, suggest a moderate negative correlation (**correlation coefficient**: -0.547 and **sig**: 0.100); which, however, is not statistically significant. This result, also suggests that new code of better quality tends to reduce the risk of generating interest, but this result cannot be generalized. To visualize the difference, we split the dataset into two groups (poor quality–$TD_{density} > 1.0$ and good quality–$TD_{density} < 1.0$) and provide the boxplots of IGRI–see Figure 5. Despite the difference in the mean values (2.55 for Good Quality Code vs. 7.49 for Poor Quality Code), the two samples do not differ statistically significantly. However, this could be due to the small sample size of our case study.

By synthesizing the results of RQ2.1 and RQ2.2 we can claim that new code is associated with the risk of generating interest: the more new code is inserted

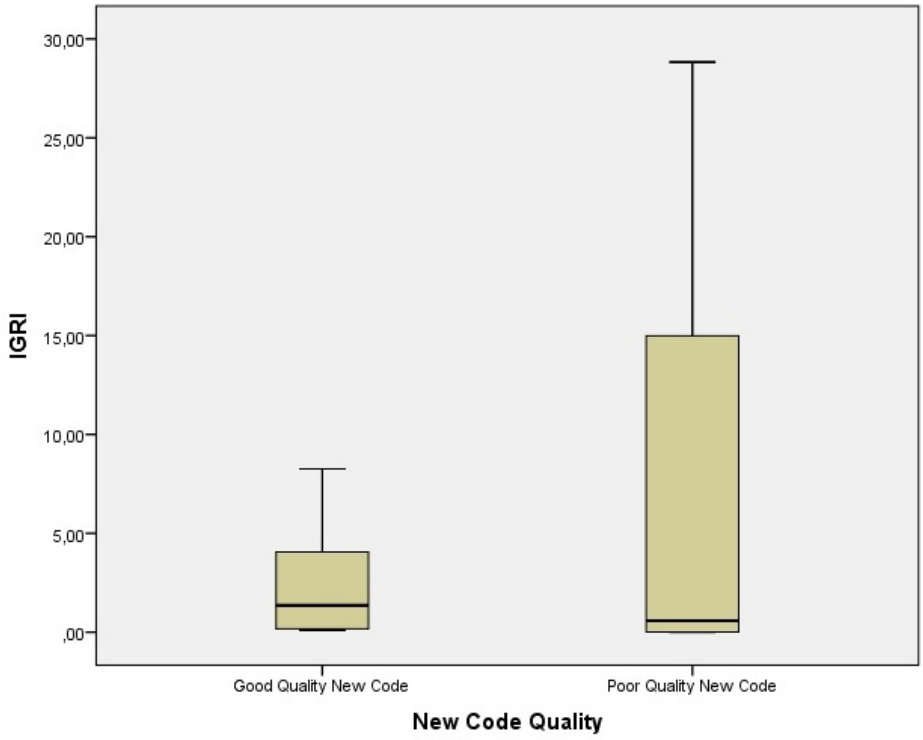

Fig. 5: Quality of New Code and Risk of Producing Interest

along evolution the lower the risk, and if this new code is "*clean*" the impact of the Technical Debt Interest Risk is further reduced. This result, complies with the literature, suggesting that clean new code reduces the amount of Technical Debt Principal along evolution [17,44]. Additionally, we emphasize that the amount of new code appears to be a more important factor for reducing the risk of producing Technical Debt Interest, compared to the quality of the code. This finding is surprising as code of better quality would be expected to decrease the risk of heavy maintenance; thus it deserves further investigation.

6 Discussion

Interpretation of Results. The findings of this study (RQ1) confirm that the proposed metric captures with sufficient accuracy the urgency of fixing problems, as it is perceived by software engineers. This result is reasonable: technical debt items with limited probability to undergo changes in the future, are naturally deemed as less urgent to fix. The same holds for items with reduced interest; software engineers are less concerned about the maintenance of artifacts that exhibit low interest (because they are simple or well-designed). The second research question of the study revealed that the risk of code packages to generate interest is negatively associated to the amount (frequency and extent) of new code introduced into them along evolution. New code is often of better quality than the existing

code base; thus, the more new code is added in each revision, the lower the risk of incurring technical debt interest.

Implications for Researchers and Software Practitioners. Prioritizing preventive maintenance tasks is a key activity in Technical Debt Management, especially for large codebases with numerous opportunities for improvement. The proposed Interest Generation Risk Importance (IGRI) captures accurately the perception of software engineers as to whether a software package should be 'refactored' to address its technical debt. We conjecture that IGRI can efficiently prioritize software artifacts at other levels of granularity as well, but this is a subject of future work. A development team can systematically obtain IGRI by tracking technical debt interest (though a framework such as SDK4ED) and the frequency of changes to software artifacts. Furthermore, the preliminary evidence that new code (and especially 'better' new code) is associated with lower risk of incurring interest, highlights the importance of tracking the quality of new code. Imposing the use of Quality Gates as a means of controlling the quality of new code can naturally lower the risk of maintainability issues in the future.

7 Threats to Validity

Before concluding our study we enumerate factors which pose potential threats for the validity of the findings and the study itself.

Construct Validity refers to the extent by which the study succeeds in properly linking theory and observations. As primary construct validity threat in technical debt analysis we should acknowledge the inherent difficulties in assessing technical debt interest. Interest is defined as the additional (future) maintenance effort because of code, design or architectural inefficiencies. By definition, future maintenance cannot be anticipated neither the additional effort compared to an ideal TD-free implementation.

Reliability, which is related to the validity of conclusions, assesses whether the findings might be heavily dependent on the involved researchers and their methods. To mitigate any reliability threats we report all steps followed to obtain the dataset for the investigated research questions and provide links to the employed tools. Moreover, the employed dataset along with the variable values for the statistical analysis are available in a replication package[5].

External Validity is related to the ability of generalizing the findings beyond the particular context in which the study has been performed. The current study suffers from such threats as only one software system, written in a particular language, has been analyzed. Given the importance of technical debt prioritization, we plan to conduct further studies on the validity of IGRI in other settings.

8 Conclusions

Acknowledging that efficient prioritization of technical debt repayment is key for software sustainability, we have introduced a simple, yet effective way to estimate risk importance of technical debt items. By considering both the amount of technical debt interest as well as the probability of artifacts to undergo changes, we

[5] https://drive.google.com/drive/folders/1c2RX6KmmBCLoU-ac2uEPxc5F2NMlSgjx

have proposed the Interest Generation Risk Importance (IGRI) measure. IGRI quantifies the impact of a possible change in a specific artifact that suffers from technical debt.

The empirical validation in an industrial setting revealed that IGRI captures accurately the notion of urgency to fix issues, as perceived by software engineers. Moreover, the more new code is added to a software system, the lower the risk to generate interest, compared to already existing code. Future work can shed light into the particular characteristics of software artifacts and development practices that lead to increased risk of technical debt interest generation.

References

1. Alves, N.S., Mendes, T.S., de Mendonça, M.G., Spínola, R.O., Shull, F., Seaman, C.: Identification and management of technical debt: A systematic mapping study. Information and Software Technology **70**, 100–121 (2016)
2. Ampatzoglou, A., Ampatzoglou, A., Chatzigeorgiou, A., Avgeriou, P.: The financial aspect of managing technical debt: A systematic literature review. Information and Software Technology **64**, 52–73 (2015). DOI https://doi.org/10.1016/j.infsof.2015.04.001
3. Ampatzoglou, A., Arvanitou, E.M., Ampatzoglou, A., Avgeriou, P., Tsintzira, A.A., Chatzigeorgiou, A.: Architectural decision-making as a financial investment: An industrial case study. Information and Software Technology p. 106412 (2020)
4. Ampatzoglou, A., Michailidis, A., Sarikyriakidis, C., Ampatzoglou, A., Chatzigeorgiou, A., Avgeriou, P.: A framework for managing interest in technical debt: An industrial validation. In: Proceedings of the 2018 International Conference on Technical Debt, TechDebt '18, p. 115–124. Association for Computing Machinery, New York, NY, USA (2018). DOI 10.1145/3194164.3194175. URL `https://doi.org/10.1145/3194164.3194175`
5. Arvanitou, E.M., Ampatzoglou, A., Bibi, S., Chatzigeorgiou, A., Stamelos, I.: Monitoring technical debt in an industrial setting. In: Proceedings of the Evaluation and Assessment on Software Engineering, EASE '19, p. 123–132. Association for Computing Machinery, New York, NY, USA (2019). DOI 10.1145/3319008.3319019. URL `https://doi.org/10.1145/3319008.3319019`
6. Arvanitou, E.M., Ampatzoglou, A., Chatzigeorgiou, A., Avgeriou, P.: A method for assessing class change proneness. In: Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering, EASE'17, p. 186–195. Association for Computing Machinery, New York, NY, USA (2017). DOI 10.1145/3084226.3084239. URL `https://doi.org/10.1145/3084226.3084239`
7. Boehm, B.: Software risk management. In: European Software Engineering Conference, pp. 1–19. Springer (1989)
8. Boehm, B., Sullivan, K.: Software economics: A roadmap, the future of software engineering. In: Proceedings of the 22nd International Conference on Software Engineering, pp. 319–343 (2000). DOI https://doi.org/10.1145/336512.336584
9. Boehm, B.W.: Software risk management: principles and practices. IEEE Software **8**(1), 32–41 (1991)
10. Boehm, B.W.: Software risk management: principles and practices. IEEE software **8**(1), 32–41 (1991). DOI https://doi.org/10.1109/52.62930
11. Charalampidou, S., Arvanitou, E.M., Ampatzoglou, A., Avgeriou, P., Chatzigeorgiou, A., Stamelos, I.: Structural quality metrics as indicators of the long method bad smell: An empirical study. In: 2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 234–238. IEEE (2018)
12. Chidamber, S.R., Darcy, D.P., Kemerer, C.F.: Managerial use of metrics for object-oriented software: an exploratory analysis. IEEE Transactions on Software Engineering **24**(8), 629–639 (1998)
13. Conejero, J.M., Rodríguez-Echeverría, R., Hernández, J., Clemente, P.J., Ortiz-Caraballo, C., Jurado, E., Sánchez-Figueroa, F.: Early evaluation of technical debt impact on maintainability. Journal of Systems and Software **142**, 92 – 114 (2018). DOI https://doi.org/10.1016/j.jss.2018.04.035. URL `http://www.sciencedirect.com/science/article/pii/S0164121218300736`
14. Cunningham, W.: The wycash portfolio management system. OOPS Messenger **4**(2), 29–30 (1993). URL `http://dblp.uni-trier.de/db/journals/oopsm/oopsm4.html#Cunningham93`
15. Digkas, G., Ampatzoglou, A., Chatzigeorgiou, A., Avgeriou, P.: On the temporality of introducing code technical debt. In: 13th International Conference on the Quality of Information and Communications Technology (QUATIC 2020). Springer (2020)

16. Digkas, G., Chatzigeorgiou, A.N., Ampatzoglou, A., Avgeriou, P.C.: Can clean new code reduce technical debt density. IEEE Transactions on Software Engineering (2020)
17. Digkas, G., Lungu, M., Chatzigeorgiou, A., Avgeriou, P.: The evolution of technical debt in the apache ecosystem. In: European Conference on Software Architecture, pp. 51–66. Springer (2017). DOI https://doi.org/10.1007/978-3-319-65831-5_4
18. Gelenbe, E., Zhang, Y.: Performance optimization with energy packets. IEEE Systems Journal **13**(4), 3770–3780 (2019)
19. Harrington, H.J.: Poor-Quality Cost: Implementing, Understanding, and Using the Cost of Poor Quality. CRC Press (1987)
20. Jankovic, M., Kehagias, D., Siavvas, M., Tsoukalas, D., Chatzigeorgiou, A.: The sdk4ed approach to software quality optimization and interplay calculation
21. Kazman, R., Cai, Y., Mo, R., Feng, Q., Xiao, L., Haziyev, S., Fedak, V., Shapochka, A.: A case study in locating the architectural roots of technical debt. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 2, pp. 179–188 (2015)
22. Kouros, P., Chaikalis, T., Arvanitou, E.M., Chatzigeorgiou, A., Ampatzoglou, A., Amanatidis, T.: Jcaliper: search-based technical debt management. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, pp. 1721–1730 (2019)
23. Li, W., Henry, S.: Object-oriented metrics that predict maintainability. Journal of Systems and Software **23**(2), 111 – 122 (1993). DOI https://doi.org/10.1016/0164-1212(93)90077-B. URL http://www.sciencedirect.com/science/article/pii/016412129390077B. Object-Oriented Software
24. Li, Z., Avgeriou, P., Liang, P.: A systematic mapping study on technical debt and its management. Journal of Systems and Software **101**, 193–220 (2015)
25. Marantos, C., Lamprakos, C.P., Tsoutsouras, V., Siozios, K., Soudris, D.: Towards plug&play smart thermostats inspired by reinforcement learning. In: Proceedings of the Workshop on INTelligent Embedded Systems Architectures and Applications, pp. 39–44 (2018)
26. Martin, R.C.: Clean code: a handbook of agile software craftsmanship. Pearson Education (2009)
27. Nikolaidis, N., Digkas, G., Ampatzoglou, A., Chatzigeorgiou, A.: Reusing code from stackoverflow: the effect on technical debt. In: 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA'19). IEEE (2019)
28. Papadopoulos, L., Marantos, C., Digkas, G., Ampatzoglou, A., Chatzigeorgiou, A., Soudris, D.: Interrelations between software quality metrics, performance and energy consumption in embedded applications. In: Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems, pp. 62–65 (2018)
29. Riaz, M., Mendes, E., Tempero, E.: A systematic review of software maintainability prediction and metrics. In: 2009 3rd International Symposium on Empirical Software Engineering and Measurement, pp. 367–377. IEEE (2009). DOI https://doi.org/10.1109/ESEM.2009.5314233
30. Runeson, P., Host, M., Rainer, A., Regnell, B.: Case study research in software engineering: Guidelines and examples. John Wiley & Sons (2012)
31. Sas, D., Avgeriou, P., Fontana, F.A.: Investigating instability architectural smells evolution: an exploratory case study. In: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 557–567. IEEE (2019)
32. Seaman, C., Guo, Y.: Chapter 2 - measuring and monitoring technical debt. pp. 25 – 46. Elsevier (2011). DOI https://doi.org/10.1016/B978-0-12-385512-1.00002-5. URL http://www.sciencedirect.com/science/article/pii/B9780123855121000025
33. Siavvas, M., Gelenbe, E.: Optimum Checkpointing for Long-running Programs. In: 15th China-Europe International Symposium on Software Engineering Education (2019)
34. Siavvas, M., Gelenbe, E.: Optimum interval for application-level checkpoints. In: 2019 6th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/2019 5th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom), pp. 145–150. IEEE (2019)
35. Siavvas, M., Gelenbe, E., Kehagias, D., Tzovaras, D.: Static analysis-based approaches for secure software development. In: International ISCIS Security Workshop, pp. 142–157. Springer, Cham (2018)
36. Siavvas, M., Marantos, C., Papadopoulos, L., Kehagias, D., Soudris, D., Tzovaras, D.: On the relationship between software security and energy consumption. In: 15th China-Europe International Symposium on Software Engineering Education (2019)
37. Siavvas, M., Tsoukalas, D., Jankovic, M., Kehagias, D., Chatzigeorgiou, A., Tzovaras, D., Anicic, N., Gelenbe, E.: An empirical evaluation of the relationship between technical debt and software security. In: 9th International Conference on Information Society and Technology (ICIST), vol. 2019 (2019)
38. Skiada, P., Ampatzoglou, A., Arvanitou, E.M., Chatzigeorgiou, A., Stamelos, I.: Exploring the relationship between software modularity and technical debt. In: 2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 404–407. IEEE (2018)

39. Society, I.C.: 1061-1998: IEEE Standard for a Software Quality Metrics Methodology. IEEE (2009)
40. Sommerville, I.: Software Engineering, 9th edn. Addison-Wesley Publishing Company, USA (2010)
41. Tsimpourlas, F., Papadopoulos, L., Bartsokas, A., Soudris, D.: A design space exploration framework for convolutional neural networks implemented on edge devices. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **37**(11), 2212–2221 (2018)
42. Tsintzira, A., Ampatzoglou, A., Matei, O., Ampatzoglou, A., Chatzigeorgiou, A., Heb, R.: Technical debt quantification through metrics: An industrial validation. In: 15th China-Europe International Symposium on Software Engineering Education (CEISEE' 19) (2019)
43. Xygkis, A., Soudris, D., Papadopoulos, L., Yous, S., Moloney, D.: Efficient winograd-based convolution kernel implementation on edge devices. In: 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC), pp. 1–6. IEEE (2018)
44. Zabardast, E., Gonzalez-Huerta, J., Šmite, D.: Refactoring, bug fixing, and new development effect on technical debt: An industrial case study. In: 46th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2020). IEEE (2020)