SN Computer Science (2021) 2:12
https://doi.org/10.1007/s42979-020-00406-6

ORIGINAL RESEARCH

The Risk of Generating Technical Debt Interest: A Case Study

Georgios Digkas[1] · Apostolos Ampatzoglou[2] · Alexander Chatzigeorgiou[2] · Paris Avgeriou[1] · Oliviu Matei[3] · Robert Heb[3]

Received: 1 July 2020 / Accepted: 13 November 2020
© The Author(s) 2020

Abstract
Technical Debt (TD) interest refers to the extra maintenance costs incurred by the very existence of TD items in a system. The generation of TD interest can make or break a system: too little interest and the effect of TD is negligible; too much interest and the system becomes unsustainable. In this paper, we consider the generation of interest as a risk and present a metric to quantify this risk. Subsequently, we validate this metric in two ways. First, we explore whether the metric can be effectively used to prioritize TD remediation. Second, we investigate if adding new code reduces the risk of interest generation. The results of the study suggest that: (a) the proposed risk management metric 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

Introduction

Technical Debt is a software engineering metaphor that 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 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 maintenance costs [18]; such costs are termed ***TD Interest*** and include 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*** [28].

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 is in such cases unnecessary. On the contrary, modules with TD that are often maintained can cause severe overhead when performing future changes. Thus, we consider the generation of interest as a *risk* that threatens software maintainability.

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

This article is part of the topical collection "Interaction between Energy Consumption, Quality of Service, Reliability and Security, Maintainability of Computer Systems and Network" guest edited by Erol Gelenbe.

✉ Georgios Digkas
g.digkas@rug.nl

Apostolos Ampatzoglou
a.ampatzoglou@uom.edu.gr

Alexander Chatzigeorgiou
achat@uom.edu.gr

Paris Avgeriou
paris@cs.rug.nl

Oliviu Matei
oliviu.matei@holisun.com

Robert Heb
robert.heb@holisun.com

[1] Institute of Mathematics and Computer Science, University of Groningen, Groningen, Netherlands

[2] Department of Applied Informatics, University of Macedonia, Macedonia, Greece

[3] Holisun SRL, Baia Mare, Romania

Published online: 16 December 2020

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.

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 [22]. Artifacts that pose a higher risk to generate TD interest would be more urgent for refactoring 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 [23], and it tends to be preferable from the developers' side, as a means to control the amount of technical debt in the system [5].

The research work 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 trade-offs between optimizations applied to improve Technical Debt, Security, and Energy dissipation in software intensive systems. Furthermore, the SDK4ED platform 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 [32], to improve performance [30, 31] and energy consumption [37], and to improve software maintainability [3, 11] is a non-trivial task. Research has proved the existence of interrelations between these qualities [25, 33, 34] rendering the extraction of the best possible sequence of software refactoring subject to a Multi-Criteria Decision-Making (MCDM) analysis which has been implemented in the SDK4ED platform.

The rest of the paper is organized as follows. In "Related Work and Background Information", we present: (a) related work on technical debt prioritization; (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 are introduced in "Assessing the Risk of Generating Interest". In "Case Study Design", we present the empirical design through which we explore the two aforementioned usage scenarios of the metric. In "Results", we answer the research questions. In "Discussion", we discuss the main findings, whereas in "Threats to Validity", the main threats to validity. Finally, in "Conclusions", we conclude the paper.

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

Related Work and Background Information

In this section, we present related work and background information necessary for understanding this study. In particular, in "Technical Debt Prioritization", we present related work: i.e., studies on technical debt prioritization; whereas in "Software Risk Management", we discuss background concepts from the software risk management literature.

Technical Debt Prioritization

The process of TD prioritization ranks identified TD items, according to certain predefined rules to support deciding which TD items should be repaid first and which TD items can be tolerated until later releases [22]. According to Li et al. [22], 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, according to Seaman and Guo [28], TD prioritization can be performed, either based on Technical Debt principal, Technical Debt interest, or Technical Debt interest probability. With respect to approach, existing methods for TD prioritization can be categorized into four main classes.

- The first class uses cost/benefit analysis, suggesting that if resolving a TD item can yield a higher benefit than cost, then this TD item should be repaid. TD items with higher cost/benefit ratios of repayment should be repaid first [29].
- The second class suggests that TD items that are more costly to resolve should be repaid first [20].
- The third class uses portfolio management. In these approaches, TD items along with other new functionalities and bugs are considered as risks and investment opportunities (i.e., assets). "The goal of portfolio management is to select the asset set that can maximize the return on investment or minimize the investment risk [17]"
- The final class suggests that TD items incurring the higher interest should be repaid first [28].

Software Risk Management

Risk management is a software engineering practice (involving processes, methods, and tools) that: (a) assesses

Fig. 1 Risk assessment matrix

Likelihood	Consequence: Negligible 1	Minor 2	Moderate 3	Major 4	Catastrophic 5
5 Almost certain	Moderate 5	High 10	Extreme 15	Extreme 20	Extreme 25
4 Likely	Moderate 4	High 8	High 12	Extreme 16	Extreme 20
3 Possible	Low 3	Moderate 6	High 9	High 12	Extreme 15
2 Unlikely	Low 2	Moderate 4	Moderate 6	High 8	High 10
1 Rare	Low 1	Low 2	Low 3	Moderate 4	Moderate 5

continuously what can go wrong (risks); (b) determines what risks are important to deal with; and (c) implements strategies to deal with those risks [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 [36]): 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.

Categorical risk assessment. According to Sommerville [36], 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 Fig. 1.

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.

Assessing the Risk of Generating Interest

This section focuses on tailoring the concepts covered in "Software Risk Management" 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], which can be formulated as:

$$\text{IGRI} = \text{Interest} \times \text{Interest Probability}. \quad (1)$$

The rest of this section describes the way that TD Interest and TD Interest Probability are calculated.

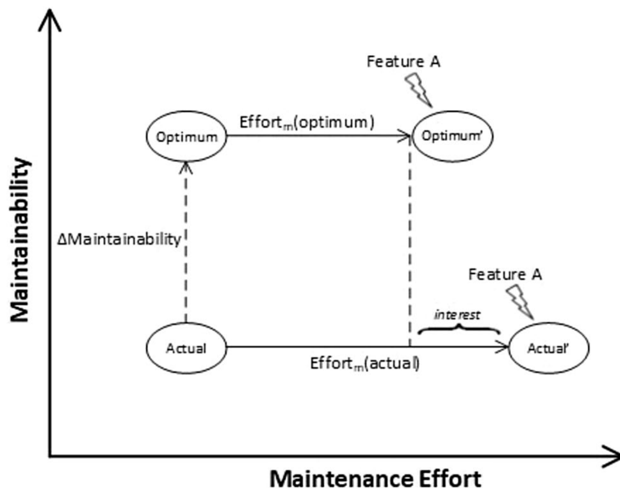

Fig. 2 Increased maintenance effort for technical debt items [3]

Technical Debt Interest

In this study, we calculate Technical Debt Interest based on the FITTED framework [2]. The estimation of Technical Debt Interest is a challenging endeavor as interest refers to the "additional" maintenance costs, incurred by inefficiencies in software. 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 FITTED methodology, which has been proposed [2] and empirically validated in our previous work [4, 38], attempts to calculate interest by estimating the "sub-optimality" of any given software artifact. FITTED assumes that any software artifact (or an entire system) has an actual implementation, and a hypothetical optimal one (in terms of maintainability). Maintaining the optimal system would require less effort than maintaining the actual system (see Fig. 2).

Despite the fact that a system can by no means be characterized as globally optimal, based solely on the optimization of some structural characteristics, several studies in the area of multi-objective software optimization aim at extracting an optimal sequence of refactoring operations that improve the software quality [24]. As shown in Fig. 2, adding a new feature A to the optimal system would need a certain effort, noted as Effort(optimum), whereas adding the same feature to the actual system necessitates a larger effort, noted as Effort (actual). The difference between these two efforts represents the Technical Debt Interest that is accumulated during this maintenance activity. The overarching assumption of FITTED is that maintenance effort is inversely proportional to the maintainability of the system (or of an individual artifact)—see Eq. 2:

$$\text{Effort} = \alpha \times (1/\text{maintainability}). \tag{2}$$

Given Eq. 2, the ratio of the maintenance effort for the optimal system (which is unknown) over the effort for the actual system can be expressed as:

$$\frac{\text{Effort}_{\text{opt}}}{\text{Effort}_{\text{act}}} = \frac{\text{maintainability}_{\text{act}}}{\text{maintainability}_{\text{opt}}}. \tag{3}$$

For convenience, we call the ratio in the right-hand side of Eq. 3 *Maintainability Level* of the actual artifact, as it expressed its relative quality compared to its hypothetical optimal implementation. Thus, the effort for maintaining the optimal system can be expressed as:

$$\text{Effort}_{\text{opt}} = \text{Effort}_{\text{act}} \times \text{MaintainabilityLevel}_{\text{act}}. \tag{4}$$

Finally, based on its definition, Technical Debt Interest can be calculated using the difference between the actual and the optimal effort, as follows:

$$\text{TD Interest} = \text{Effort}_{\text{act}} \times (1 - \text{MaintainabilityLevel}_{\text{act}}). \tag{5}$$

Maintainability. Although no single function can capture all aspects of quality, for the sake of simplicity, we assume that the hypothetical *'optimal'* system is the one that optimizes a certain fitness function assessing the quality of software (e.g., in terms of complexity, cohesion, coupling, etc. or any aggregate form of selected qualities). To enable the calculation of the aforementioned maintainability level, we first identify a set of artifacts (e.g., classes, packages, and systems [4]) that can be considered (structurally) similar, i.e., in terms of lines of code, number of methods, cognitive complexity, etc. Next, we calculate the optimal value of selected metrics among the set of similar artifacts. These best metric scores are assumed to characterize the hypothetical *'optimal'* which the artifact under study could potentially reach. A simplified example is outlined in Fig. 3.

Then, we calculate the average ratio of the metric score of the artifact under study, compared to the optimal value, yielding its maintainability level. The metrics that we have selected to use in our study for quantifying maintainability (see Table 1) belong to well-known metric suites [12, 21]. The metric selection was based on a secondary study by Riaz et al. [26], which reported on a systematic literature review (SLR) aimed at summarizing software metrics that can be used as maintainability predictors.

Maintenance Effort. Since the evolution of software cannot be predicted under normal circumstances, it is not possible to foresee what kind of modifications (e.g., bug fixes, introduction of new features, refactoring, etc.) will be made in a system during future releases. Hence, we follow a simple, yet relatively reasonable approach, and base our assessment of future maintenance effort on historical data. In particular, to consider past effort spent on maintenance

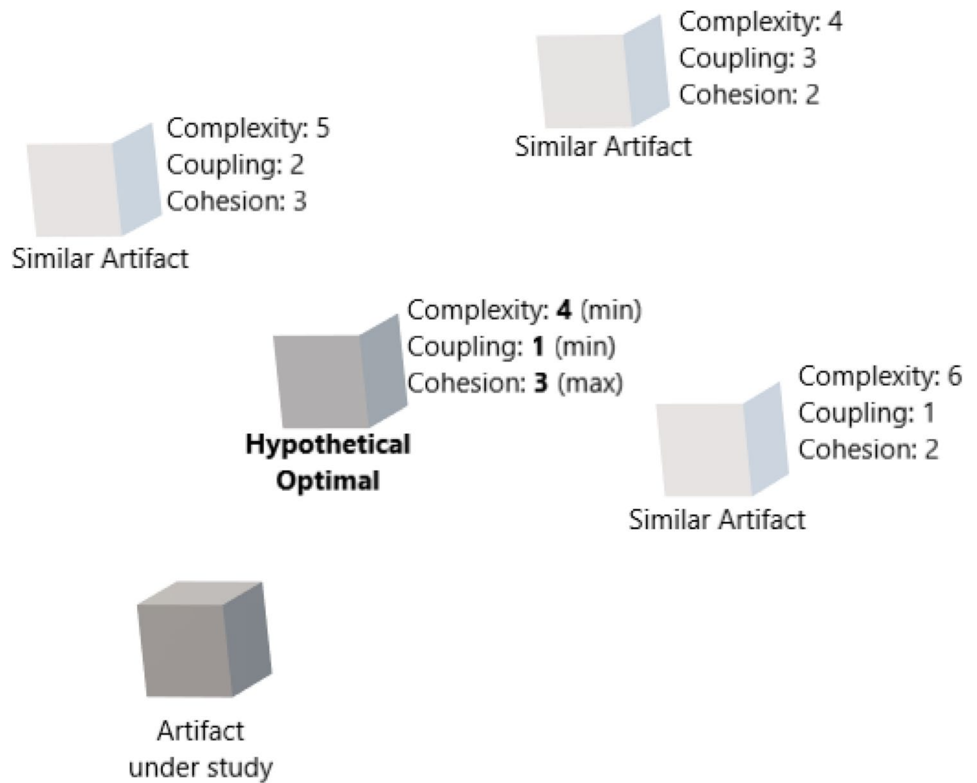

Fig. 3 Notion of hypothetical optimal among similar artifacts

Table 1 Maintainability proxy metrics

Property	Metric	Description
Inheritance (Inh)	DIT	Depth of Inheritance Tree: Inheritance level number, 0 for the root class.
	NOCC	Number of Children Classes: Number of direct sub-classes that the class has.
Coupling (Cpl)	MPC	Message Passing Coupling: Number of send statements defined in the class.
	RFC	Response for a Class: Number of local methods plus the number of methods called by class methods.
	DAC	Data Abstraction Coupling: Number of abstract types defined in the class.
Cohesion (Coh)	LCOM	Lack of Cohesion of Methods: Number of disjoint sets of methods in the class.
Complexity (Com)	CC	Cyclomatic Complexity: Average cyclomatic complexity of methods in the class.
	WMPC	Weighted MethoWeighted
Size (Size)	SIZE1	Lines of Code: Number of semicolons in the class.
	SIZE2	Number of Properties: Number of attributes and methods in the class.

activities for each artifact, we record the average lines of code added/deleted/modified between all pairs of successive versions of a system (code churn). Consequently, we project this average maintenance effort per version to future releases of the analyzed artifact. This strategy has been used in a variety of studies on software evolution [13, 19].

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 there is a relation between Maintenance Effort and PCCC, 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.

Case Study Design

The case study is designed and reported based on the linear-analytic structure as described by Runeson et al. [27]. This section presents the study design in detail.

Research Goals and Questions

The goal of this study, as mentioned in the Introduction section, is twofold: (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 that correspond to these two goals:

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 "Introduction", 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.

Table 2 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

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 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. The system consists of six main modules, managing the following entities: (a) fiches of progress, (b) actions to be taken, (c) documents involved in ISO quality control, (d) planning, (e) useful information, and (f) milestones.

Data Collection

To answer the aforementioned research questions, we have performed the following process. 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 2. The first column corresponds to the name of the

[2] https://sdk4ed.eu.

[3] The selection process was as follows. First, we sorted the packages by IGRI, and 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.

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

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 (five 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 mapping identified technical debt issues to the line range of these methods. Note that we use the concept of $\mathbf{TD}_{\text{density}}$, which is the technical debt of these methods normalized over their size in lines of code. $TD_{density}$ 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 in the following phases and individual steps:

Phase 1: Retrieval of commits

1. First, the git history for the project under study is retrieved from its master branch.
2. All commits are sorted to form a time-series of revisions that have been performed on the source code. 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 reduce the computation time, a filtering step is applied by ignoring transitions between successive commits that do not involve any changes to Java files.

Phase 2: Mapping of technical debt issues to methods

To map the identified technical debt issues to the class methods of each revision, we perform the following steps:

1. First, for each revision, we retrieve all technical debt issues by performing the corresponding query to the SonarQube database.
2. Next, we map the identified technical debt issues to the methods of the corresponding revision. This is performed by matching the line in which each technical debt issue is reported by SonarQube with the method containing that line.

Phase 3: Tracking new methods

We identify the introduction of new methods and the associated $TD_{density}$ as follows:

1. For the new files of each revision (obtained from git history), we obtain their representation in the form of an Abstract Syntax Tree (AST)[4]. For each new file, we extract all its methods from the AST representation and then tag all these methods as new.
2. For the modified files of each revision, we track new methods in each transition with the help of the Gumtree Spoon AST Diff tool[5].

Phase 4: Calculating the contribution of new methods to the change in the system’s $TD_{density}$

Finally, we need to calculate, for each revision in the system’s history, the contribution of new methods to the change of the system’s $TD_{density}$. Let us consider a transition from revision *t-1* to revision *t*. The contribution of new methods to the change in the $TD_{density}$ of the system is obtained with the following formula:

Contribution of new methods

$$\Delta \mathrm{TD}_{\mathrm{density}}(\mathrm{new}) = \frac{\mathrm{TD}_{t-1} + \mathrm{TD}_{\mathrm{new}(t)}}{\mathrm{LOC}_{t-1} + \mathrm{LOC}_{\mathrm{new}(t)}} - \mathrm{TD}_{\mathrm{density}}(t-1). \tag{6}$$

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

[4] The AST is obtained through the Eclipse Java Development Tools (JDT).

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

[V4] Average LoC added as new code in the history of the package

[V5] Average contribution of new code in the $TD_{density}$ of the package.

Data 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], and for RQ2.1, we use 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 contributions) and provided additional analysis.

Results

In this section, we present the results of the case study, organized by research question. In particular, in "Ability of IGRI to Prioritize TD Artifacts", we present the results on the ability of IGRI to predict the risk of software packages to produce high interest. Subsequently, in "Relation of IGRI and New Code", we use the newly proposed index to assess the contribution of new code to the risk of producing technical debt interest.

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, according to the 1061–1998 IEEE Standard for Software Metrics [35]), in Fig. 4, we present a scatter plot.

As it can be observed from Fig. 4, 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

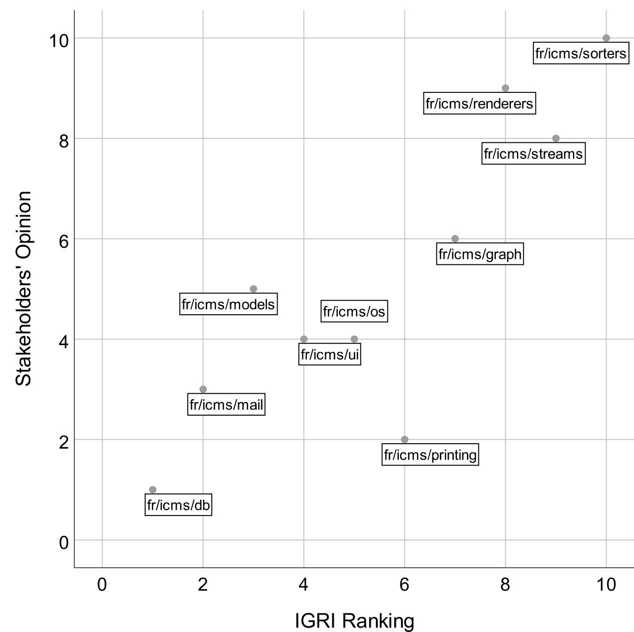

Fig. 4 Risk ranking consistency of IGRI and perception of stakeholders

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 risks 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 [35] the ranking that an experienced practitioner would provide to packages in terms of risk to generate Technical Debt Interest.

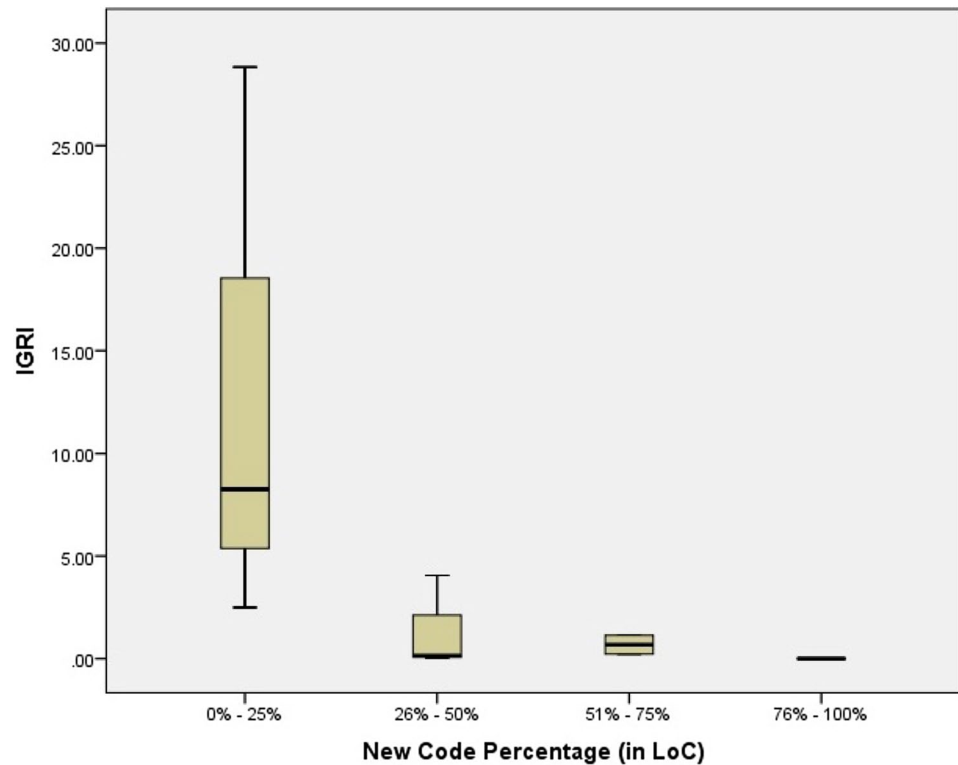

Fig. 5 Percentage of new code and risk of producing interest

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.

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 [16, 39]. 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 exists 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 Fig. 5, 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 Fig. 5, 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).

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 Fig. 6. 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 related to the risk of generating interest. In the general case, the more new code is inserted along evolution, the lower the risk, and if this new code is "*clean*",

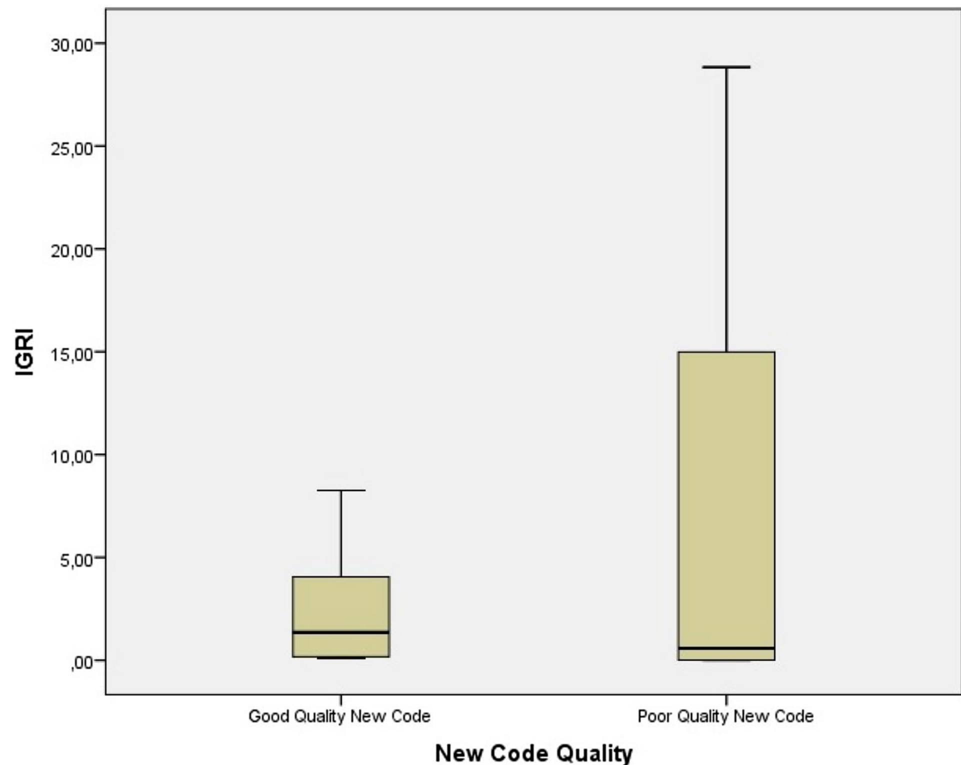

Fig. 6 Quality of new code and risk of producing interest

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 [16, 39]. 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.

Discussion

Interpretation of Results. The findings of this study (RQ1) confirm that the proposed metric captures with sufficient accuracy and 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 with 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.

Threats to Validity

In this section, we discuss threats to the validity of the study, including threats to construct, external validity, and reliability. The study does not aim at establishing cause-and-effect relations, and is thus not concerned with internal validity.

Construct Validity reflects how far the examined phenomenon is connected to the intended objectives. 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 reflects whether the study has been conducted and reported in a way that others can replicate it and reach the same results. 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 is available in a replication package[6].

External Validity is related to the ability of generalizing the findings to other settings, e.g., other software projects, other programming languages, and possibly other technical debt tools. 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.

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 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.

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

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

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Alves NS, Mendes TS, de Mendonça MG, Spínola RO, Shull F, Seaman C. Identification and management of technical debt: a systematic mapping study. Inf Softw Technol. 2016;70:100–21.
2. Ampatzoglou A, Ampatzoglou A, Chatzigeorgiou A, Avgeriou P. The financial aspect of managing technical debt: a systematic literature review. Inf Softw Technol. 2015;64:52–73. https://doi.org/10.1016/j.infsof.2015.04.001.
3. 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; 2018. p. 115–124.
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. https://doi.org/10.1145/3194164.3194175.
5. Arvanitou EM, 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. https://doi.org/10.1145/3319008.3319019.
6. Arvanitou EM, 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. https://doi.org/10.1145/3084226.3084239.
7. Boehm B. Software risk management. In: European Software Engineering Conference. Springer; 1989. p. 1–19.
8. Boehm B, Sullivan K. Software economics: a roadmap, the future of software engineering. In: Proceedings of the 22nd International Conference on Software Engineering; 2000. p. 319–343. https://doi.org/10.1145/336512.336584
9. Boehm BW. Software risk management: principles and practices. IEEE Softw. 1991;8(1):32–41.
10. Boehm BW. Software risk management: principles and practices. IEEE Softw. 1991;8(1):32–41. https://doi.org/10.1109/52.62930.
11. Charalampidou S, Arvanitou EM, 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); 2018. p. 234–238. IEEE
12. Chidamber SR, Darcy DP, Kemerer CF. Managerial use of metrics for object-oriented software: an exploratory analysis. IEEE Trans Softw Eng. 1998;24(8):629–39.
13. Conejero JM, Rodríguez-Echeverría R, Hernández J, Clemente PJ, Ortiz-Caraballo C, Jurado E, Sánchez-Figueroa F. Early evaluation of technical debt impact on maintainability. J Syst Softw. 2018;142:92–114. https://doi.org/10.1016/j.jss.2018.04.035http://www.sciencedirect.com/science/article/pii/S0164121218300736.
14. Cunningham W. The wycash portfolio management system. OOPS Messenger. 1993;4(2):29–30 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, Lungu M, Chatzigeorgiou A, Avgeriou P. The evolution of technical debt in the apache ecosystem. In: European Conference on Software Architecture. Springer; 2017. p. 51–66. https://doi.org/10.1007/978-3-319-65831-5_4
17. Guo Y, Seaman C. A portfolio approach to technical debt management. In: Proceedings of the 2nd Workshop on Managing Technical Debt, MTD '11, p. 31–34. Association for Computing Machinery, New York, NY, USA; 2011. https://doi.org/10.1145/1985362.1985370.
18. Harrington HJ. Poor-quality cost: implementing, understanding, and using the cost of poor quality. Boca Raton: CRC Press; 1987.
19. 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; 2015. vol. 2, p. 179–188.
20. Letouzey JL. The sqale method for evaluating technical debt. In: 2012 Third International Workshop on Managing Technical Debt (MTD). 2012; p. 31–36. IEEE. https://doi.org/10.1109/MTD.2012.6225997
21. Li W, Henry S. Object-oriented metrics that predict maintainability. J Syst Softw. 1993;23(2):111–22. https://doi.org/10.1016/0164-1212(93)90077-B. http://www.sciencedirect.com/science/article/pii/016412129390077B. Object-Oriented Software.
22. Li Z, Avgeriou P, Liang P. A systematic mapping study on technical debt and its management. J Syst Softw. 2015;101:193–220.
23. Martin RC. Clean code: a handbook of agile software craftsmanship. London: Pearson Education; 2009.
24. Ouni A, Kessentini M, Sahraoui H. Multiobjective optimization for software refactoring and evolution. In: Advances in Computers, vol. 94. Elsevier; 2014. p. 103–167. https://doi.org/10.1016/B978-0-12-800161-5.00004-9
25. 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; 2018. p. 62–65.
26. 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; 2009. p. 367–377. IEEE. https://doi.org/10.1109/ESEM.2009.5314233
27. Runeson P, Host M, Rainer A, Regnell B. Case study research in software engineering: Guidelines and examples. Hoboken: Wiley; 2012.
28. Seaman C, Guo Y. Chapter 2—measuring and monitoring technical debt. Elsevier; 2011. p. 25 – 46. https://doi.org/10.1016/B978-0-12-385512-1.00002-5. http://www.sciencedirect.com/science/article/pii/B9780123855121000025
29. Seaman C, Guo Y, Zazworka N, Shull F, Izurieta C, Cai Y, Vetrò A. Using technical debt data in decision making: potential decision approaches. In: 2012 Third International Workshop on Managing Technical Debt (MTD); 2012. pp. 45–48. IEEE. https://doi.org/10.1109/MTD.2012.6225999
30. Siavvas M, Gelenbe E. Optimum Checkpointing for Long-running Programs. In: 15th China-Europe International Symposium on Software Engineering Education; 2019.
31. 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); 2019. pp. 145–150. IEEE.
32. Siavvas M, Gelenbe E, Kehagias D, Tzovaras D. Static analysis-based approaches for secure software development. In: International ISCIS Security Workshop. Springer, Cham; 2018. pp. 142–157.
33. 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.
34. 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.
35. Society IC. 1061–1998: IEEE standard for a software quality metrics methodology. IEEE; 2009.
36. Sommerville I. Software engineering. 9th ed. Boston: Addison-Wesley Publishing Company; 2010.
37. Tsimpourlas F, Papadopoulos L, Bartsokas A, Soudris D. A design space exploration framework for convolutional neural networks implemented on edge devices. IEEE Trans Comput Aided Des Integr Circuits Syst. 2018;37(11):2212–21.
38. 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.
39. 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.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.