Empirical Software Engineering (2024) 29:25
https://doi.org/10.1007/s10664-023-10412-w

A metrics-based approach for selecting among various refactoring candidates

Nikolaos Nikolaidis[1] · Nikolaos Mittas[2] · Apostolos Ampatzoglou[1] · Daniel Feitosa[3] · Alexander Chatzigeorgiou[1]

Accepted: 18 October 2023 / Published online: 16 December 2023

Abstract

Refactoring is the most prominent way of repaying Technical Debt and improving software maintainability. Despite the acknowledgement of refactorings as a state-of-practice technique (both by industry and academia), refactoring-based quality optimizations are debatable due to three important concerns: (a) the impact of a refactoring on quality is not always positive; (b) the list of available refactoring candidates is usually vast, restricting developers from applying all suggestions; and (c) there is no empirical evidence on which parameters are related to positive refactoring impact on quality. To alleviate these concerns, we reuse a benchmark (constructed in a previous study) of real-world refactorings having either a positive or negative impact on quality; and we explore the parameters (structural characteristics of classes) affecting the impact of the refactoring. Based on the findings, we propose a metrics-based approach for guiding practitioners on how to prioritize refactoring candidates. The results of the study suggest that classes with high coupling and large size should be given priority, since they tend to have a positive impact on technical debt.

Keywords Technical Debt · Refactoring · Empirical Quantitative Analysis · Interest · Principal

Communicated by: Bibi Stamatia, Maxime Cordy, Bowen Xu, Xiaofei Xie

✉ Apostolos Ampatzoglou
a.ampatzoglou@uom.edu.gr

Nikolaos Nikolaidis
nnikolaidis@uom.edu.gr

Nikolaos Mittas
nmittas@chem.ihu.gr

Daniel Feitosa
d.feitosa@rug.nl

Alexander Chatzigeorgiou
achat@uom.edu.gr

1 Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece

2 Department of Chemistry, International Hellenic University, Kavala, Greece

3 Bernoulli Institute for Mathematics, Computer Science and Artificial Intelligence, University of Groningen, Groningen, The Netherlands

1 Introduction

The *Technical Debt* (TD) metaphor expresses in monetary terms the effort that a software development team saves or "*borrows*", by opting for a "*quicker*" but "*non-optimal*" development approach in terms of quality—implying that consequently, interest will have to be paid. *TD Interest* expresses the additional effort that teams will need to pay during software maintenance, because of the presence of inefficiencies, while the cost to resolve these inefficiencies is called *TD Principal*. Large amounts of TD Interest are an important concern for software development teams, as it essentially describes the future cost of '*sweeping problems under the carpet*' that are neither evident in the short term and are not easy to quantify or even accurately predict in the future (Arvanitou et al. 2020). *TD Management* (TDM) is the process that systematically assesses TD, monitors its evolution, and when necessary, suggests actions for reducing the amount of TD Principal, which in turn, is expected to limit the amount of TD Interest. Current empirical evidence suggests that following the laws of software evolution (Lehman et al. 1997), TD Principal usually increases in absolute value as a system grows; however, effective TDM can lead to a reduction of TD density (TD normalized over the total lines of code), as it is evident in software projects with well-defined quality assurance processes (Digkas et al. 2022).

According to the literature, there are both re-active and pro-active approaches for TD reduction. On the one hand, the re-active strategy, which is the most common in industry (Smiari et al. 2022), refers to the application of refactoring to purposefully eliminate code, design or architectural smells and implementation flaws that may exist. On the other hand, the pro-active approaches, that seem to be appealing to individual practitioners (Ampatzoglou et al. 2019), but are not yet well-established, yield for the definition of Quality Gates that impose the merging in the main development branch, only of code that is "*cleaner*" compared to the average quality in the code base (Digkas et al. 2022). Other, stricter policies for quality control impose the "*zero-bug*" policy in the main branch: allowing only code commits with limited (under a defined threshold) or zero violations against a pre-defined set of rules (Falessi et al. 2017a). In this study, we focus on the re-active strategy, i.e., the application of refactoring, which despite being well-accepted and recognized as a useful and practical solution, is haunted by empirical uncertainty, and practical limitations, as outlined in Fig. 1, and discussed below:

Size of the Solution Space: Identifying refactoring candidates can be performed either manually (Ge et al. 2012) or with tool support (Campbell and Papapetrou 2013; Tsantalis et al. 2008; Sharma et al. 2016). For the latter case (which is the most prominent in the industry (Ivers et al. 2022)), a practical problem is that the list of refactoring candidates is usually so long that the developer cannot cope with efficiently processing it. Thus, there is a need for *an approach and a tool implementation that could automatically prioritize the refactoring candidates*. ***Uncertainty of Refactoring Impact***: Several studies have investigated the impact of refactorings on various aspects of software quality, but the results are contradictory: i.e., identify cases that the refactoring has a positive impact, and others that the refactoring is neutral, or even has a negative impact (see Section 2). To this end, it is important to provide *an approach that can pin-point to refactoring candidates that will have a positive impact on quality after their application* (TD Principal and TD Interest), relying on information available before the refactoring. ***Refactoring Impact Parameters***: Given the above, the developers need to apply an automated prioritization approach, relying on the available pieces of information: (a) the type of the refactoring (e.g., Extract Method, Extract Class, etc.); and (b) the class or

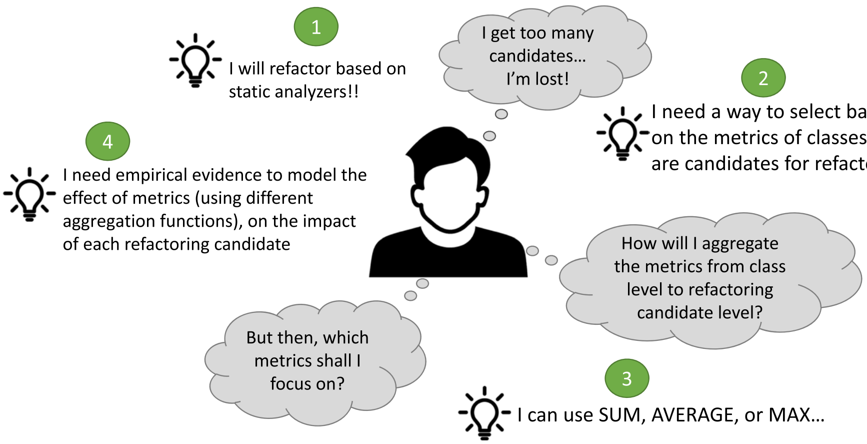


Fig. 1 Study Motivation and Problem Statement

the set of classes that constitute the candidate for refactoring. According to practitioners (Smiari et al. 2022; Ampatzoglou et al. 2019; Ivers et al. 2022) refactoring prioritization would be more efficient, based on "*where*" the refactoring is applied, given that "*design hotspots*" (i.e., parts of code with particularly low quality) should receive more attention. To this end, there is a need for an *approach that prioritizes refactoring candidates, based on the characteristics of classes (Refactoring Impact Parameters) that would be involved in the refactoring process*. This need leads to a counter problem: "*How does someone aggregate the characteristics of a group of classes to a unified unit of analysis*? In our case: from the class-level to the refactoring candidate-level".

To alleviate this problem, we provide empirical evidence on which structural metrics (Refactoring Impact Parameters) can be inspected (after being aggregated to the Refactoring Candidate level) before the application of a refactoring, to increase the probability of selecting a beneficial refactoring, i.e., having a positive impact in terms of TD. To achieve this goal, we need to rely on past (historical) data, i.e., cases in which a refactoring had a positive or a non-positive (neutral or negative) impact on TD. To construct this dataset, we reuse the process proposed by Nikolaidis et al. (Nikolaidis et al. 2023), who identified isolated (from other maintenance activities) refactoring applications, along the history of various software projects, relying on the mechanism of labelling *Pull Requests* (PRs). The process for constructing the dataset is presented in detail in Section 4.3. Upon the construction of the dataset, we apply a thorough experimental setup via the fitting of Generalized Linear Mixed Models (GLMMs), using Refactoring Impact Parameters (a set of structural metrics for the classes that are Refactoring Candidates) as independent variables; whereas as response variable a binary value for the impact of the refactoring (positive or non-positive) is used. A similar process of using mixed effects models has also been followed in previous studies on TDM (Nikolaidis et al. 2023), given their appropriateness for the nested nature of data: i.e., Refactoring Candidates are grouped by the Project they belong to. The rest of the paper is organized as follows: In Section 2 we discuss related work, while in Section 3 we briefly outline the employed quantification approach, which forms the backbone of this quantitative study. The design of our case study is presented in Section 4, along with the corresponding research questions. The results are presented and discussed in Section 5, the implications to both practitioners and researchers are presented in Section 6. We identify threats to the validity of the study in Section 7; and finally, we conclude the paper in Section 8.

2 Related Work

In this section, we present existing studies and background information for this paper. In Section 2.1, we focus on the effect of refactorings on software quality. This section aims at showcasing the need for this study, i.e., by presenting the controversy of empirical findings on the impact of refactorings: some cases positive, others having a negative effect. In Section 2.2, we present related work that shares a common study setup, in terms of unit of analysis. Thus, we present studies that analyze Pull Requests, instead of classes, commits, or projects. Finally, in Section 2.3, we present directly related work, i.e., studies that aim to estimate the impact that a refactoring could have on software quality.

2.1 Refactoring and Software Quality

Murphy-Hill et al. (Murphy-Hill et al. 2011) investigated the habits of developers in terms of refactoring and found that developers rarely perform refactoring-related activities. But when a refactoring does take place, the effect on quality is still uncertain. Various studies have observed positive and negative impacts. Kataoka et al. (Kataoka et al. 2002) evaluated the impact of the "Extract Method" and the "Extract Class" refactoring methods on a software project's maintainability, written in C++, using coupling metrics. The results indicate that refactorings magnify system maintainability from the perspective of code metrics. Stroulia and Kapoor (Stroulia and Kapoor 2001) investigated the effect on size and coupling measures after the application of refactoring and their results show that the average LOC of involved classes and coupling metrics decreased after refactoring.

On the other hand, Stroggylos and Spinellis (Stroggylos and Spinellis 2007) inspected the logs in the version control systems of four open-source software projects to extract the revisions where software refactoring had taken place. The findings reveal that, despite the expectation that a refactoring improves the quality of the software, the measurements in the examined systems show the opposite. In particular, the authors observed that code refactoring caused a slight increase in cohesion and coupling related metrics. In another study, Alshayeb (Alshayeb 2009) concluded that the application of refactoring does not necessarily improve external quality characteristics, such as adaptability, maintainability, and comprehensibility. By applying refactoring techniques, as defined by Fowler, on three software systems and measuring the effect on selected software metrics, vast discrepancies in the effect of refactoring were revealed. The author concluded that it was not possible to corroborate that software refactoring as a general practice can improve quality.

Also noteworthy is the study by Wilking et al. (Wilking et al. 2007), who conducted a controlled experiment to investigate how refactoring affects the conservation and modification of projects. The results of their experiment suggest that there is no direct effect of software refactoring leading to improved maintainability. Most of the findings of the above studies agree on the limited practical adoption of software refactoring and on a rather mixed effect on the quality of a project, at least on quality aspects that can be quantified. Moreover, the study of Moser et al. (Moser et al. 2008) approached the problem from a perspective closer to the industry. The authors examined whether refactoring increased productivity as well as code quality. To achieve that, they relied on a small industrial tool which captures the productivity and the code metrics of each developer. They found out that productivity increases after a refactoring takes place and that code metrics improve.

Al Omar et al. (AlOmar et al. 2019) analyzed a total of 1,245 commits from 3,795 Java projects to capture the effect that the developer intended to achieve through a refactoring versus the actual effect. To this end, several projects were analyzed looking for the refactorings that took place with the help of the *Refactoring Miner* (Tsantalis et al. 2022) and *ReffDiff* (Silva and Valente 2017) tools. Next, they retained only the commits where the corresponding message explicitly specified the exact quality attribute that was the target. By analyzing the before- and after- state of the code for each commit, it becomes possible to determine if the developer achieved the desired outcome through the applied refactoring. It was found out that for the quality attributes of cohesion, coupling, and complexity refactorings were able to capture the intention of the developer, but for the rest of the metrics the refactoring in the commit had not affected them.

Finally, another study with mixed results was conducted by Bois and Mens (Bois and Mens 2003). They took a different approach from the rest of the previous studies, in the sense that they based their analysis on the abstract syntax tree (AST) representation of source code. They examined the change of the metric values defined on the AST representation for different types of refactorings like Extract Method, Encapsulate Field, and Pull Up Method. The application of the examined refactorings showed positive and negative impacts on the studied metrics.

2.2 Pull Requests and Software Quality

In the literature, it is common to explore PRs to find their contributions to the quality of the submitted code or even their acceptance based on the quality of the new code. This goes to show the importance of studying PRs since they constitute a cohesive set of commits that makes a specific bug-fix, addition, or maintenance activity of a given project. Silva et al. (Silva et al. 2016) analyzed 1,722 PRs and found that 30% of the rejected PRs are due to the presence of technical debt issues. Also interesting is that the most frequently attributed rejection reason was code design, which was also identified in the study by Zou et al. (Zou et al. 2019). Zou et al. analyzed 50,000 PRs from 117 different projects to find whether the coding style affects the PRs chances from being eventually merged. This study found out that the more consistent the added code is to the already existing code base, the higher the probability of a merging the PRs. Another study analyzed PRs from the perspective of code quality for three projects, namely Spark, Kafka, and React (Karmakar et al. 2022). This study showed that the discussion of technical debt in PRs appears to be different than in other software artifacts (e.g., code comments, commits, issues, or discussion forums).

Regarding the more general software quality area, Gousios et al. (Gousios et al. 2015) conducted a large-scale survey of 749 participants, that act as integrators in many different systems to find out the factors that affect the decision of accepting or not a PRs. The code quality was the top factor that influenced the decision of the integrators, along with the testing and the alignment with the project's overall idea. The main takeaway was that both technical and social factors play a significant role in the PRs acceptance. The social aspect was found (and confirmed) to be a very important aspect in other studies as well (Moreira Soares et al. 2021), where the developer was the most important factor that influenced the chances of a PR. Finally, Lenarduzzi et al. (Lenarduzzi et al. 2021) analyzed more than 36,000 PRs from a total of 28 Java projects focusing on whether the quality of the code that is being introduced is related to the acceptance probability of the PR. In that study, the PMD tool was used to find code quality defects, and it was evident that quality did not play a key role in the acceptance or rejection of the PRs. However, it seems that certain PMD rules are indeed considered by the reviewers for the acceptance of new code. Similar results were found in other studies (Calefato et al. 2017), where the developers' trustworthiness was more important as well as the code quality and structure.

2.3 Estimating the Impact of Refactorings

There is a plethora of tools regarding refactoring recommendations. Kurbatova et al. (Kurbatova et al. 2020) proposed an approach to recommend Move Method refactorings that relies on the path-based representation of code, and they used this to train a machine learning classifier. After some evaluation, it was obvious that this approach can stand against, or even outperform in some cases, the state of the art tools. In another study, Murphy-Hill et al. (Murphy-Hill and Black 2008) stated the importance of refactoring tools and presented three new ones. These tools could help developers with Extract Method refactorings by avoiding selection errors and understanding refactoring precondition violations. These tools were also assessed by their accuracy and their user satisfaction, which was very high. Moreover, approaches have been proposed that help developers prioritize and select the most effective or profitable refactorings (Mavridis et al. 2012; Meananeatra et al. 2011). Similarly, SEMI, a tool that helps with the prioritization of Extract Method refactorings, uses a ranking approach, based on the benefits that each refactoring is going to have in the overall cohesion, based on the single responsibility principle (Charalampidou et al. 2016).

Chaparro et al. (Chaparro et al. 2014) created an approach named RIRE, which can predict the values of some structural metrics based on the refactoring that is going to take place. RIPE can calculate the impact of 12 different refactoring operations, on 11 structural metrics. Even though some of the refactoring operations have very good accuracy for some metrics, in a test case RIPE was only able to perfectly predict 38% of 8,103 metric scores. The evaluation took place in 15 Java projects and a total of 504 refactorings. A similar study, but on a smaller scale, was conducted by Kataoka et al. (Kataoka et al. 2002). They used only metrics that are related to coupling and refactoring methods that affect coupling, like Extract Method, Extract Class, and Move Method.

Moreover, Higo et al. (Higo et al. 2008), created a methodology and tool, which can recommend a refactoring based on its effect on the quality of the project. In that study, 6 metrics were used from the CK metric suite. To validate the results a real-world example was used, and the methodology was able to propose a refactoring that was used at the end. The study by Soetens and Demeyer (Soetens and Demeyer et al. 2010) analyzed the evolution of the complexity of a project. Then, by extracting the commit that explicitly stated that a refactoring was applied, it was possible to isolate the effect that it had on complexity. The most important takeaway was that many times the complexity of the project was not reduced, while at a closer look it was found that complexity was highly correlated with the type of refactoring.

This paper is the first one that goes a step further than related work, which until now examined the effect of refactorings on quality or TD. More specifically, we not only highlight the problem of the controversy of empirical findings, but we also present actionable rules, and guide the practitioners on when to apply a refactoring and when not. This is an important advancement compared to state-of-the-art, since it moves from exploratory analysis to an explanatory level, with actionable results.

3 Background Information

3.1 Software Quality

Software quality is an ambiguous term depending on the viewpoints of different stakeholders, being characterized as an "*elusive target*" (Kitchenham and Pfleeger 1996), implying that developing "*perfect quality*" software is not feasible in practice, since not all quality parameters can be optimized, either because of the associated cost or due to the inherent trade-offs among quality attributes. Thus, on the one hand, from the prespective of the developer, quality is related to the conformance of software to its specification. On the other hand, from the perspective of the user, quality is whether the software meets its purpose. In general, software quality is assessed through its inherent characteristics. Several standards have been proposed, but the most popular is ISO/IEC 25010. The first level of this ISO describes eight quality attributes, i.e., functional suitability, performance/efficiency, compatibility, reliability, usability, security, maintainability, and portability, which are further divided into several sub-characteristics. For instance, software maintainability is decomposed to: modularity, reusability, testability, analyzability, and modifiability. To assess and quantify a quality attribute (of the first or the second level) the development teams need to define or select a set of metrics, based on the development phase (Arvanitou et al. 2016), which are going to provide insights for the achieved level of software quality.

3.2 Code (Bad) Smells

The term "code smells" is used to describe parts of code or decisions that are generally associated with bad design and bad programming practices. Code smells are used to locate the places in software that could benefit from refactorings and Fowler et al. (Fowler et al. 1999) described 22 possible code smells and their associated refactorings. In contrast to bugs, smells do not cause a fault in the application but may lead to other negative consequences, impacting software maintenance and evolution. Detection of code smells has become an established method to indicate software design issues that may cause problems for further development and maintenance, and they are being considered one of the key indicators of TD (Alves et al. 2016b). SonarQube, which is one of the most frequently used tools for estimating TD, relies on 273 rules associated with code smells (for Java)—e.g., "*Boolean expressions should not be gratuitous*", "*Conditionals should start on new lines*", etc. SonarQube rules that are related to code smells are linked to code understandability, poorly written code, and coding standards. But there are more rules related to bugs, vulnerabilities, and security hotspots.

3.3 Technical Debt Quantification

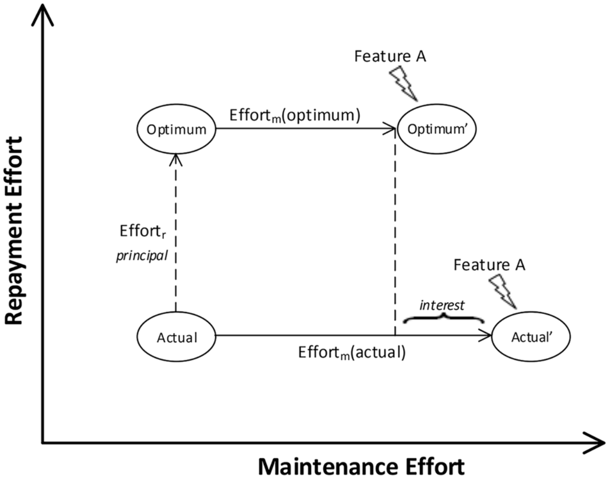
The main pillars of the TD metaphor are Principal and Interest, which are borrowed from economics. TD Principal is the effort that is required to remove inefficiencies from the current state of a software system to bring it closer to an "*optimal"* state (Ampatzoglou et al. 2015a). On the other hand, TD Interest refers to the extra development effort that is required to maintain the software, due to the presence of inefficiencies (TD Principal) (Soetens and Demeyer et al. 2010).

In Fig. 2, a hypothetical software system is depicted in a maintenance state of "*actual*". The actual quality is usually at some distance from the "*optimum*" quality: The effort required for the development team to close this quality gap, represents the TD Principal. The consequence of the existence of the principal is TD Interest, which represents the additional effort required to maintain the software in the actual state, compared to the effort that would be required if the system was of optimal quality. In other words, for the introduction of a hypothetical `Feature A` to the system, the development team would require less time, if the system had been of an optimal (or at least 'better than actual') quality (Chatzigeorgiou et al. 2015). The estimation of TD Principal is more straightforward as it is related to the identification of pre-defined inefficiencies, while the estimation of TD Interest is more challenging as it involves the anticipation of future changes and the assessment of the additional maintenance effort.

In the last decade or more, numerous ***TD Principal*** quantification tools have been proposed that estimate TD Principal, either in monetary terms or as effort (in time) to repay TD (Avgeriou et al. 2021). There have also been numerous studies about the proposed tools and their accuracy (Avgeriou et al. 2021; Li et al. 2015; Lefever et al. 2021). In this study we rely on SonarQube, which is the most frequently used tool for estimating TD Principal, according to several studies (Avgeriou et al. 2021; Alves et al. 2016a). SonarQube can capture the TD Principal by finding the code inefficiencies of the given system and calculate the required time to resolve the corresponding issues. The platform algorithm was originally based upon an adopted version of the SQALE method proposed by Letouzey (Letouzey 2012), in which a remediation index is obtained for requirements of an applicable Quality Model. Moreover, SonarQube supports more than 20 programming languages, and it performs static analysis against a specified set of rules. For the Java programming language, that interests us in this study, SonarQube version 9.7.1 checks for violations against 627 rules. These rules are divided into 4 categories based on their type, namely bug, vulnerability, code smell, and security hotspot. Finally, apart from the remediation time of each issue, there is also a severity scale (blocker, critical, major, minor, info).

For ***TD Interest***, the quantification is far more challenging, mainly due to the need to anticipate the future state of a given system. First, a system can by no means be characterized as optimal, based solely on the optimization of some structural characteristics. Second, to calculate the TD Interest, the maintenance effort to add a feature in the actual state and in a hypothetical one, would be needed; the latter cannot be calculated accurately. In our study, we adopt the FITTED approach (Ampatzoglou et al. 2015b), which has been proposed and empirically validated in our previous work (Ampatzoglou et al. 2018; Tsintzira et al. 2019). The proposed TD Interest quantification approach is based on historical data, by considering past effort spent on maintenance activities and using the average number of lines of code added between sequential releases as a maintenance effort indicator. The derivation of an "*optimal*" peer for any given class is as follows: (a) find the 5 closest neighbors (classes of the system) of the class under study, based on structural characteristics—e.g., number of methos, lines of code, number of attributes, etc., (b) based on them we develop an "*artificial*" optimum peer (i.e., being characterized by the best metric scores of peers). The distance of the class (in terms of maintenance-related metrics) determines the additional maintenance effort for that class. The FITTED methodology estimates the approximate additional maintenance effort for each class which can be turned into monetary terms by multiplying with an average wage. According to Tsintzira et al. (Tsintzira et al. 2019) the FITTED TD Interest quantification approach is correlated at the level of 0.73 to the perception of practitioners in terms of the amount of additional effort required to maintain an existing industrial system, due to the presence of TD.

Fig. 2 TD Principal and Interest Visualization (Chatzigeorgiou et al. 2015)



4 Case Study Design

4.1 Objectives & Research Questions

The goal of this study is to identify Refactoring Impact Parameters—***RIP*** (i.e., structural characteristics of the Refactoring Candidates—***RC***—before the application of the refactoring) that can assess the positive or non-positive effect of a refactoring on the values of TD Principal and/or TD Interest. By considering that quantifying the structural characteristics at the level of RC (calculated by aggregating the class-level metrics socres) is not a trivial problem in the software engineering domain (due to the presence of various aggregation functions), we decompose the goal on two main research questions:

> **RQ$_1$**: *Can the selection of a function to aggregate metric scores (from the class- to the RC-level) affect the ability to identify the impact of refactoring activities on TD Principal and TD Interest?*

When a developer gets a refactoring suggestion, he/she is not expected to change the code of only one file. Thus, to be able to compare RCs of different sizes (in terms of involved classes), there is a need of aggregating the metric scores, from the class to the RC level. In the software engineering literature (Ampatzoglou et al. 2020), the most used aggregation functions are ***Mean***, ***Sum***, and ***Max***, each one yielding for a different interpretation. For example, using ***Sum*** as an aggregation function takes into consideration the number of classes to be refactored, ***Max*** focuses only on the worst-case scenario: i.e., the worst class among those to be refactored, whereas ***Mean*** is not discriminating between large and small RCs, and it does not focus on extreme metric scores. In RQ$_1$ we investigate if different aggregation functions can lead to different factors that affect the impact of refactoring on TD Principal and TD Interest.

> **RQ$_2$**: *Which Refactoring Impact Parameters (at the level of RC) can affect the impact of the refactoring on TD?*

In RQ_2, we aim to model the impact of a refactoring on TD Principal and/or Interest, based on the aggregated metrics scores (RIP) of a RC. In other words, we attempt to identify relations between specific RIPs and positive impact of refactoring, i.e., "*Which metrics should have high/low values so as for the application of RC to have better chances for a positive impact on TD?*". To answer this question, as comprehensively as possible, we first examine the effect of RIPs to the impact of applying the RC on TD Principal ($RQ_{2.1}$), then to the impact on TD Interest ($RQ_{2.2}$), and finally to the impact on both—i.e., positive impact on both Principal and Interest ($RQ_{2.3}$).

4.2 Case Selection and Units of Analysis

The cases of this study are open-source software (OSS) projects that are subject to systematic maintenance, including the application of refactoring applications. All of the selected projects can be found in Table 1, along with some basic characteristics to initially describe the sample. It becomes evident that 10 out of the 15 projects are part of the Apache ecosystem, since as an OSS development organization, has a reputation for high quality projects, emphasizing on process and quality improvement, while having long maintained projects. The five remaining projects are from other organizations, in an attempt for more generalizable results. To select the most fitting cases and ensure their homogeneity, but also their diversity, we navigated and selected projects from the most frequently maintained and popular projects (from the "explore" tab of GitHub), while also have defined the following criteria:

- [C1] The OSS project is ***written in Java and uses Maven*** to ensure that the project can be analyzed. We note that the FITTED tool for calculating TD Interest is available only for Java code, and SonarQube can provide better results if it is part of the build process (since it takes into account rules based on the exact Java version that is being used).
- [C2] The OSS project is ***currently under development;*** thus, is still maintained. This criterion aims at ensuring that the projects included in the analysis are still undergoing development; therefore, the studied practices are not outdated; increasing the chances for identifying refactorings.
- [C3] The OSS project has ***more than 250 closed pull requests*** to have enough data points for each project. The closed pull requests can be either merged or not, and from the merged ones we do not expect all of them to be labeled as refactorings. Since we could not find any threshold for the number of PRs in the literature, we have intuitively set 250 as a threshold, by examining the number of PRs of OSS, as well as to ensure that we have enough data per project for our analysis.
- [C4] The OSS project ***uses labeling for refactorings in pull requests***, which is one of the most important criteria, as our study requires that a PR is labeled as "*refactoring*" to use it.

The study is a multiple case study, in the sense that from each project, multiple PRs labelled as refactorings (units of analysis) have been identified. Each applied refactoring (before its application) is considered as an RC that can be assessed as having a positive or non-positive impact on quality (in terms of TD Principal, TD Interest or both), by analyzing the project before and after the PR.

Table 1 Selected Projects

Project	Commits	PRs	Size (#LOC)	Size (# Classes)
antlr/antlr4	9106	1816	98,653	1453
Netflix/conductor	3127	1533	74,443	578
DataDog/dd-trace-java	10,889	4340	171,760	3762
apache/dolphinscheduler	7698	6867	150,826	2027
apache/doris	9387	11,745	403,813	3909
apache/druid	12,534	9058	1,032,837	9276
apache/dubbo	6491	5716	191,728	2976
apache/incubator-seatunnel	3039	2596	95,338	1700
provectus/kafka-ui	1642	1948	48,124	353
apache/pinot	9916	8371	432,607	4405
apache/pulsar	11,396	13,243	567,649	5532
apache/rocketmq	7993	2901	180,792	1823
apache/skywalking	7532	4626	104,817	2089
apache/streampipes	10,027	554	130,938	2344
uima-uimaj	7736	265	372,629	2209

4.3 Data Collection

The data collection for this study has been organized around the need to identify the refactorings that have been applied along software evolution to mine units of analysis. Tools such as *Refactoring Miner* (Tsantalis et al. 2022) can identify past refactoring activities (and have been adopted in previous related work (Nikolaidis et al. 2022)) but are not fitting to the goals of this study. In particular, the application of a refactoring is not always the main and only intention of a developer in a commit (e.g., the developer might commit a feature addition, along with a small refactoring), or a refactoring might be spread in subsequent commits. To create a dataset with changes only aiming at "*pure*" refactoring (to avoid construct validity), we rely on information that can be retrieved by studying PRs: an approach that has already been used in various studies (see Section 2.2) (Silva et al. 2016; Zou et al. 2019; Karmakar et al. 2022; Gousios et al. 2015, 2017; Moreira Soares et al. 2021; Lenarduzzi et al. 2021). In large projects that take full advantage of collaborative development environments, PRs are commonly used to submit groups changes, serving a common goal. PRs allow to contribute one or more commits for a specific functionality or change, which then must be reviewed before being merged. Because of the controlling nature of this mechanism, it is common to allow contributions to the production / main branch, only using PRs and disable the direct commit (also known as branch protection) (Hastings and Walcott 2022). The intention of a PR is usually denoted by attaching labels (like keywords) to a PR, and these labels can be customized per project and, although optional, there are some common practices on larger projects as they promote organization (Zhang et al. 2023). To construct the dataset for this study, we focused on PRs that are tagged with a label explicitly stating that a refactoring has been performed. This approach will ensure that our dataset contains changes in files, for which the refactoring was the main change that the developer wanted to achieve. To develop our dataset, we filtered refactoring-related PRs, and then we assessed the code quality in terms of TD (Principal and Interest), before and after the PR merge—characterizing the refactoring as having a positive or non-positive impact on TD Principal, on TD Interest, on both. These steps are illustrated in Fig. 3 and can be split into the following phases:

Phase 1 First, we had to extract information from the GitHub repository of each project. The two main pieces of information that we were interested in were the subset of PRs that we will need to analyze, and the list of the changed Java files. For the PRs, we retained only the closed PRs that had a specific label, designating that this PR contains a refactoring. So, we used the GitHub API to get all the PRs and filter them accordingly. From the GitHub API we were also able to retrieve, for each PR that interests us, the previous and merged commit along with the changed Java file. The main endpoints that were used are the following:

https://api.github.com/repos/<username>/<project>/pulls?state=closed&per_page=100&page=1

https://api.github.com/repos/<username>/<project>/commits/<commit-hash>

The first one retrieves all the closed PRs and the second one retrieves more information about the merged PR (i.e., the previous commit, and the changed files). To filter and organize our results we created a script that can be found online.[1]

Phase 2 After completing the list of the PRs that concerned refactoring (commits before and after the merge), we can analyze the code. So, we automated the process by checking out a specific commit each time and starting the code analysis. As part of the analysis, we calculated TD Principal and TD Interest. For TD Principal, we used SonarQube (Campbell and Papapetrou 2013), whereas for TD Interest we used FITTED (Ampatzoglou et al. 2015b). Finally, as RIP, we assessed several maintainability-related parameters, by calculating 9 structural metrics for the ***commit before the application of the refactoring***. The selected metrics (see Table 2) have been indicated by previous studies (Riaz et al. 2009a; Van Koten and Gray 2006; Zhou and Xu 2008) as the optimal maintainability predictors. To calculate the metric scores, we used Metrics Calculator,[2] a well-tested and stable tool for calculating quality metrics for Java code.

After recording the data, we have developed a dataset, with the following variables. The complete dataset can, for replication purposes, be found online.[3]

[V1] TD Principal Before Refactoring
[V2] TD Principal After Refactoring
[V3] Impact of Refactoring on TD Principal (binary)
[V4] TD Interest Before Refactoring
[V5] TD Interest After Refactoring
[V6] Impact of Refactoring on TD Interest (binary)
[V7] Impact of Refactoring on TD (V3 AND V6)
[V8–V32] RIP aggregated by SUM, MEAN, and MAX (9 RIP * 3 aggregation functions) Before Refactoring

4.4 Data Analysis

To answer the RQs posed in this study, a specialized modelling technique belonging to the broad category of Mixed Effects Models (MEMs) was adopted. MEMs are a general class

[1] https://github.com/nikosnikolaidis/github-pr-labels-filechanges

[2] https://github.com/dimizisis/metrics_calculator

[3] https://users.uom.gr/~a.ampatzoglou/aux_material/refactoring_preds.xlsx

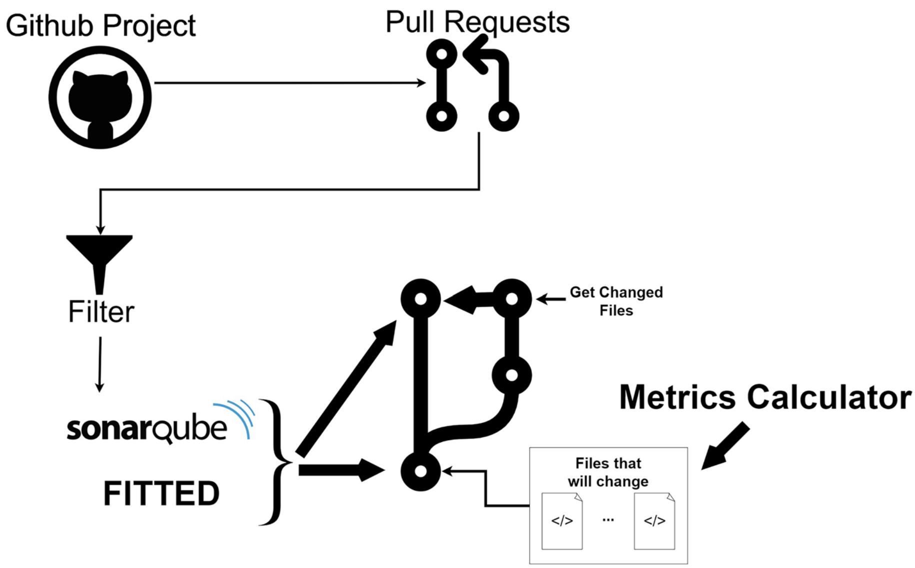


Fig. 3 Data Collection Flow

of inferential statistics methodologies, that can be grasped as an extension of the traditional Generalized Linear Models (GLMs) allowing the investigation of two types of effects, called the fixed and the random effects, on a response variable via the building of a unified single model. MEMs are useful in complicated experimental setups, in which the same units of analysis are measured multiple times and / or they are naturally grouped into a hierarchical structure. These two types of experimental designs (repeated measures and hierarchical or nested designs) pose significant barriers to the inferential statistics mechanisms, since in both cases, the assumption of the independence of observations is evidently violated. As the main objective of this study is the investigation of the effect of RIPs on the refactoring impact, it is essential to take into consideration the nested structure of the experimental setup, since multiple units of analysis (in our case: RCs) are nested into the same case (in our case: OSS projects), i.e., not being independent to each other. Hence, the two-level inherent hierarchy of the collected data and the dependency of the units of analysis that were grouped into nested factors (RC are nested within OSS Projects) were the main reasons for taking advantage of the robust MEMs rather than other statistical hypothesis testing procedures, since they provide an advanced mechanism for the incorporation of the so-called random effects and the modeling of the expected variance at different levels of hierarchy.

With respect to the response variable, the main research pillar focuses on the examination of the effect of RIPs on the refactoring impact on TD Principal and TD Interest. For this reason, two dichotomous (or binary) variables, namely [V3] (Eq. 1) and [V6] (Eq. 2) were defined, indicating whether refactoring activities were associated to a positive or non-positive impact on TD Principal and TD Interest, respectively. The basis for the categorization of refactoring activities into positive and non-positive groups was the quantification of TD Principal [V1, V2] and TD Interest [V4, V5] before and after the application of the refactoring. At this point, we must emphasize that due to the qualitative nature of the response variable (i.e., refactoring impact is a dichotomous variable with two levels (positive/non-positive)), we based the inferential process on a specific type of

Table 2 Refactoring Impact Parameters

Metric	Metric Suite	Description
CC	McCabe	Complexity based on the number of decisions in the source code
WMC	Chidamber & Kemerer	Weighted methods per class (Number of methods)
DIT	Chidamber & Kemerer	Depth of inheritance tree
LCOM	Chidamber & Kemerer	Lack of cohesion in methods (LCOM1)
NOCC	Chidamber & Kemerer	Number of Class Children
CBO	Chidamber & Kemerer	Coupling due to Method invocation, inheritance, exception handling, method parameters, field access is considered
MPC	Li & Henry	Message-passing coupling (Number of Distinct Methods Called)
SIZE1	Li & Henry	Lines of code (LOC)
SIZE2	Li & Henry	Number of properties

MEMs, namely the *Generalized Linear Mixed Models* (GLMMs) enabling the examination of a binary response through the usage of a *logit link function*. The logit link function $g(\bullet) = log(p/(1-p))$ is defined as the natural logarithm of the odds for success, where p is the probability of a successful refactoring activity.

$$V3 = \begin{cases} Positive, & \text{if} TDPrincipal_{Before} > TDPrincipal_{After} \\ Non-positive, & \text{otherwise} \end{cases} \quad (1)$$

$$V6 = \begin{cases} Positive, & \text{if} TDInterest_{Before} > TDInterest_{After} \\ Non-positive, & \text{otherwise} \end{cases} \quad (2)$$

Regarding the fixed effects (independent variables) that may affect the outcome of the response variable (refactoring impact), we have used 9 predefined quality metrics (Table 2). Since the RIPs were evaluated on a lower level of hierarchy (class- or file-level) compared to the response variable (RC-level), there is an imperative need for the aggregation of class-level metrics at the higher level (RC). For this reason, we investigated the effect of three aggregation mechanisms (***Mean***, ***Sum*** and ***Max***) on the response variable, with a strong focus on providing directions to practitioners about the most appropriate one for guiding their decision-making (RQ_1). Next, we modeled the probability of an RC having a positive impact on TD Principal ($RQ_{2.1}$) and TD Interest ($RQ_{2.2}$) as a function of the aggregation function (see RQ_1) and RIPs (fixed effects). Finally, for $RQ_{2.3}$, we followed a similar approach after the creation of a new response variable: [V7], labeling a given RC as positive, if and only if, refactorings were successful in terms of decreasing both TD Principal and TD Interest. In RQs, we controlled the variance decomposition, due to the nested structure of the experimental setup, by the application of MEMs.

5 Results

In this section we present the results of this study, organized by research question. As a first step, we investigated the distribution for the response and the independent variables to derive meaningful conclusions concerning the characteristics of the unknown population. The contingency table (Table 3) displays the marginal and joint distributions of the indicator variables [V3] and [V6] that classify RCs as Non-positive / Positive in terms of TD Principal (columns) and TD Interest (rows), for a total set of 434 refactoring candidates. The nested rows can be interpreted as follows: (a) the first nested row shows the absolute frequency of the observations in each intersection (e.g., 269 corresponds to cases with non-positive effect on both Interest and Principal); (b) the second nested row corresponds to the percentage on the aforementioned number to the total of the row (e.g., 269 corresponds to 87% of the cases with non-positive impact on Interest); and (c) the third nested row corresponds to the percentage of the aforementioned number to the total of the column (e.g., 269 stands to 100% of the cases with non-positive impact on Principal).

The marginal distributions display that in most cases, refactoring resulted in a Non-positive impact on TD Principal (row: Total) and TD Interest (column: Total). More importantly, the inspection of the joint distribution reveals that the refactoring activities that led to a Non-positive impact on TD Principal ($N = 269$), resulted in a Non-positive impact on TD Interest for most of the cases (87.3%). Furthermore, the Positive impact of refactoring on TD Principal is primarily associated to Positive impact on TD Interest, as well. This result is considered intuitive in the sense that TD Principal and TD Interest are not orthogonal concepts, but (similarly to economics) are related (Ampatzoglou et al. 2020). Additionally, the fact that there are cases of RC with Positive impact on TD Principal, but not on TD Interest (~ 13% of the sample) can be attributed to the fact that some TD issues identified by SonarQube (rule-based identification) are unrelated to structural aspects, but rather on styling or conventions conformance (Falessi et al. 2017b). On the other hand, we can observe that all structural improvements captured by the Positive impact on TD Interest are also reflected on the Positive impact on TD Principal (0% of Positive impact on TD Interest and Non-positive on TD Principal), validating that SonarQube assesses also structural properties through the rule violations (Falessi et al. 2017b).

Regarding the characteristics of the distributions for the set of RIPs, recorded through metrics, in Table 4, we summarize their main central tendency and variation measures after the application of each aggregation function (***Mean***, ***Sum***, ***Max***). The descriptive statistics, along with the indicative examination of the histograms (Fig. 4) computed by the ***Sum*** aggregation function, bring to light the heavily right-skewed distributions for the RIP scores, accompanied by the presence of extreme outlying points.

5.1 Aggregating the results from class to refactoring candidate level (RQ$_1$)

In Table 5, we summarize the overall results derived from the fitting of each GLMM that constitute the basis for further inferential purposes and decision-making related to RQs. A first interesting remark concerns the perfect agreement within the experimental findings conducted for the identification of the most appropriate aggregation function for evaluating a composite metric from class-level to the RC-level in the case of TD Principal. More specifically, a total number of 7 out of 9 structural metrics presented a statistically significant effect ($p < 0.05$) on the response variable irrespective of the applied aggregation function. In addition, there was noted a perfect agreement concerning the results related to the identification of RC-level metrics that did not present a statistically significant effect on refactoring impact on TD Principal. In contrast, the above general finding does not hold for the experiments regarding TD Interest, since, despite

Table 3 Joint Distribution of Refactoring Impact on TD Principal and TD Interest

		Principal		Total
		Non-positive	Positive	
Interest	Non-positive	269	39	308
		87.3%	12.7%	100%
		100%	23.6%	71%
	Positive	0	126	126
		0%	100%	100%
		0%	76.4%	29%
Total		269	165	434
		62%	38%	100%

the reasonable high agreement among the three aggregation functions, inconsistent outcomes for two specific cases are observed. In this regard, the utilization of the ***Mean*** aggregator did not reveal a statistically significant effect on the response variable for the total set of GLMMs, whereas the ***Sum*** and ***Max*** aggregation schemas designated the significant effect of CC and DIT on the refactoring impact on TD Interest. Regarding the uniform positive impact of the refactoring on TD Principal and Interest, we can observe that MPC can be an important RIP using all aggregation functions, whereas CBO only when using MAX, and SIZE1 (i.e., lines of code) only when using MAX and MEAN.

> *The use of different aggregation functions is an irrelevant factor if the quality assurance team is interested only in the monitoring of TD Principal. When TD Interest comes into consideration, MAX appears as the optimal choice in the sense that it is easier to inspect and pin-point more RIPs.*

5.2 Factors affecting the impact of refactoring on TD (RQ_2)

To gain deeper insights into how the aggregated RIPs may affect the impact of RCs, we performed an exploratory data analysis through visualization techniques. Due to space limitations, we illustrate the boxplots and violin plots for the set of metrics aggregated by the ***Sum*** function for the experimental setups of both TD Principal (Fig. 5) and TD Interest (Fig. 6). The examination of the distributions for the case of TD Principal provides empirical evidence that most of the RIPs can be considered as important, since they affect the impact of the refactoring, deserving further investigation. For example, refactorings with a Positive impact were associated with higher CC, WMC, LCOM, MPC, CBO, SIZE1 and SIZE2 scores compared to refactorings with a Non-positive impact.

In contrast, apart from a minority of cases (e.g., CC and DIT metrics), there are no obvious differences in the shapes of the distributions between the Non-positive and Positive groups representing the impact of refactoring activities on TD Interest. Moreover, the nature of the association between CC and the impact on TD Interest seem to be different compared to TD Principal, since refactorings with a Positive impact on TD Interest present lower CC values compared with refactorings with a Non-positive impact. A possible interpretation for this is the fact that in TD Interest calculation, metrics scores do not participate as actual values, but as distances from the scores of neighboring classes. In that sense, a refactoring that lowers the complexity of a class with high CC, might alter its '*neighborhood*'', comparing

Table 4 Descriptive Statistics of Aggregated RIP (Metrics Before Refactoring)

Aggregation	Variable	M	SD	Median	Min	Max
Mean	CC	2.46	1.96	2.00	0.00	17.38
	WMC	17.25	24.48	10.00	0.00	327.00
	DIT	1.22	1.64	0.89	0.00	12.50
	LCOM	636.15	2974.31	57.42	0.00	53,301.00
	MPC	69.12	133.23	32.45	0.00	1940.00
	NOCC	1.92	10.90	0.00	0.00	140.00
	CBO	10.55	16.14	6.34	0.00	213.00
	SIZE1	287.73	458.15	152.82	0.00	5922.00
	SIZE2	26.65	35.23	16.00	0.00	424.00
Sum	CC	34.66	89.11	6.49	0.00	777.80
	WMC	215.22	541.21	42.00	0.00	5656.00
	DIT	23.47	85.28	2.00	0.00	930.00
	LCOM	6233.75	19,880.50	204.50	0.00	160,388.00
	MPC	838.01	2223.97	124.50	0.00	24,955.00
	NOCC	16.95	61.39	0.00	0.00	368.00
	CBO	150.96	386.01	22.00	0.00	3764.00
	SIZE1	3686.05	9191.08	651.50	0.00	86,481.00
	SIZE2	323.69	805.88	69.00	0.00	8762.00
Max	CC	4.95	4.74	3.00	0.00	32.00
	WMC	49.50	77.97	19.50	0.00	444.00
	DIT	3.24	4.77	1.00	0.00	32.00
	LCOM	3957.70	12,891.83	136.00	0.00	87,735.00
	MPC	184.42	354.60	68.00	0.00	2029.00
	NOCC	13.64	58.02	0.00	0.00	346.00
	CBO	25.55	40.77	11.00	0.00	232.00
	SIZE1	905.26	1782.12	305.00	0.00	13,434.00
	SIZE2	70.85	102.25	31.00	0.00	561.00

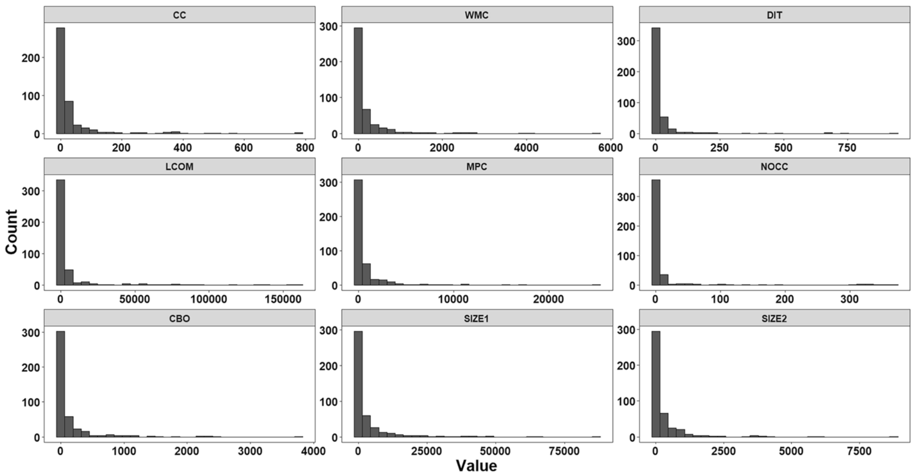


Fig. 4 Distributions of Refactoring Impact Parameters (***Sum*** aggregation)

Table 5 Results of GLMMs for TD Principal and TD Interest

Variable	Principal			Interest			Principal & Interest		
	Mean	Sum	Max	Mean	Sum	Max	Mean	Sum	Max
CC	✓	✓	✓	✗	✓	✓	✗	✗	✗
WMC	✓	✓	✓	✗	✗	✗	✗	✗	✗
DIT	✗	✗	✗	✗	✓	✓	✗	✗	✗
LCOM	✓	✓	✓	✗	✗	✗	✗	✗	✗
MPC	✓	✓	✓	✗	✗	✗	✓	✓	✓
NOCC	✗	✗	✗	✗	✗	✗	✗	✗	✗
CBO	✓	✓	✓	✗	✗	✗	✗	✗	✓
SIZE1	✓	✓	✓	✗	✗	✗	✓	✗	✓
SIZE2	✓	✓	✓	✗	✗	✗	✗	✗	✗

✗non-significant

✓significant at 0.05

them to classes with lower levels of CC. This phenomenon cannot heavily apply to low CC classes, which cannot significantly deviate from their original score (and change neighborhood). An indicator for this assumption is the fact that the value of CC can change significantly by applying the '*Replace Conditional with Polymorphism*', if the number of branches of the conditional statements is high, leading to a drastic decrease in CC (from the number of branches to zero)—being a sensitive metric (Arvanitou et al. 2016).

After the identification of the RIPs that presented a statistically significant effect on the refactoring impact on ***TD Principal***, we proceeded to the parameter estimation for this subset of metrics. Table 6 summarizes the estimated parameters of GLMMs along with their $p-$ values for TD Principal. Concerning the interpretation of the estimated parameters of GLMMs (i.e., row *Estimate*), the positive sign of an independent variable indicates that the likelihood of a Positive impact of refactoring increases, as the value of the RIP increases. In other words, RCs with higher values of RIP scores are more likely to guarantee a beneficial refactoring application. Since *Odds Ratio* (OR) in GLMMs provide an intuitively appealing and straightforward interpretation regarding the effect of changes in a predictor on the response variable, we computed the ORs of each RIP from the fitted models (Table 6, row *OR*). In our case, an $OR > 1$ indicates that a positive impact of refactoring activities is more likely to occur as the RIP score increases. Based on this simple interpretation rule, the total set of seven RIPs identified as statistically significant predictors seem to positively affect the outcome of a refactoring opportunity in TD Principal. As an example, an RC whose aggregate CC metric via the ***Mean*** aggregation function is twice as much as the CC metric of another RC, is associated with a change in the odds of a positive refactoring impact by a factor of 1.48 (or 48% increase). This finding can be considered intuitive and suggests that design hotspots with low quality (excessive metric scores) in terms of coupling, lack of cohesion, complexity, and size are more probable to undergo a refactoring leading to a positive effect on TD Principal.

A similar analysis process was followed for the closer examination of the subset of aggregated RC-level RIPs that presented a significant impact on **TD Interest**, namely CC and DIT (see Table 7). A first interesting remark concerns CC, which as explained through the violin charts has a significant effect on the refactoring impact on both TD Principal and TD Interest, but in an inverse direction. The interpretation for this controversy has already

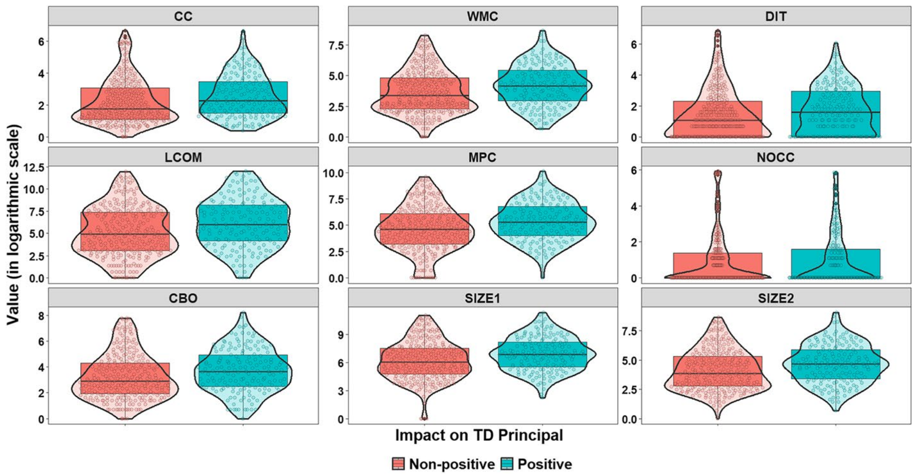


Fig. 5 Distributions of Metrics (using ***Sum***) for Refactorings with Non-positive / Positive Impact on Principal

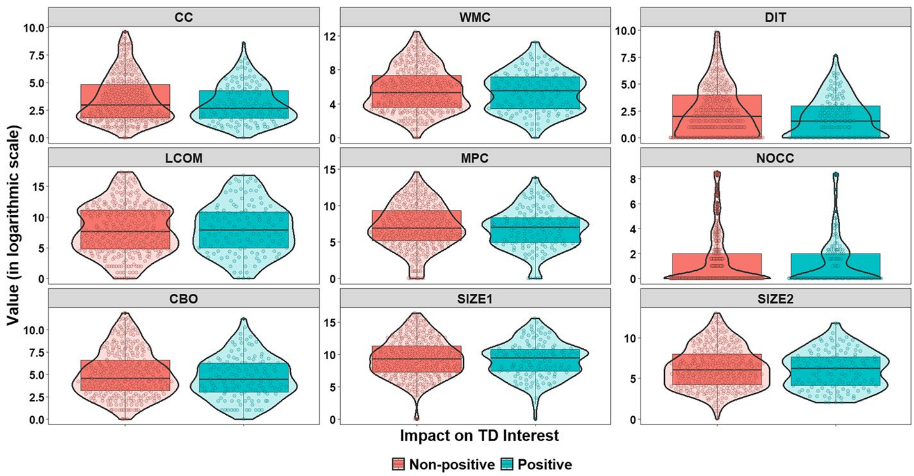


Fig. 6 Distributions of Metrics (using ***Sum***) for Refactorings with Non-positive / Positive Impact on Interest

been discussed before. Regarding, DIT the negative relation to TD Interest is intuitive, in the sense that the lower the aggregate DIT values are, the lower the use of inheritance. Given the fact that most of the Fowler refactorings (Fowler and Beck 1999) yield for the introduction of inheritance to benefit from polymorphism, we can anticipate classes outside inheritance trees presenting the largest room for beneficial refactoring application. For example, the estimated ORs ($OR < 1$) evaluated by the GLMMs fitted through the usage of the ***Sum*** and ***Max*** aggregators designate that RCs whose aggregate DIT score is twice as much as the DIT of another RC, have about 0.20 times less odds of undergoing a refactoring having a Positive impact on TD Interest.

Table 6 GLMMs Estimated Parameters for Significant RIPs (TD Principal)

Aggregation	Model	CC	WMC	LCOM	MPC	CBO	SIZE1	SIZE2
Mean	Estimate	0.392	0.195	0.092	0.194	0.198	0.205	0.194
	SE	0.187	0.096	0.041	0.071	0.105	0.075	0.099
	OR	1.479	1.215	1.096	1.214	1.219	1.227	1.214
	p	0.036	0.041	0.024	0.006	0.049	0.006	0.049
Sum	Estimate	0.122	0.115	0.077	0.119	0.102	0.125	0.118
	SE	0.058	0.048	0.030	0.042	0.048	0.044	0.049
	OR	1.129	1.121	1.080	1.126	1.107	1.133	1.125
	p	0.037	0.017	0.011	0.005	0.033	0.004	0.016
Max	Estimate	0.353	0.171	0.076	0.178	0.198	0.170	0.184
	SE	0.123	0.068	0.032	0.058	0.077	0.058	0.072
	OR	1.423	1.186	1.079	1.194	1.219	1.185	1.202
	p	0.004	0.012	0.018	0.002	0.010	0.003	0.011

Technical Debt The last part of the experimental setup is related to the identification of RIPs that affect the probability of an RC having a positive impact on both TD Principal and TD Interest. The results are presented in Table 8. An interesting finding from this analysis is that the two RIPs that have been identified as significant for TD Interest have not been qualified as significant for the intersection of TD Principal and TD Interest. A possible explanation is that high CC increases the chances for a Positive impact on TD Principal but decreases the chances for a Positive impact on TD Interest. Out of the RIPs that have a significant effect on the impact of the refactoring either on TD Principal or TD Interest, MPC, CBO, and SIZE1 appear to be able to affect the impact of the refactoring on both pillars of TD. For all cases the Estimate is positive, which follows the rationale for the design hotspots.

RCs involving classes with excessive MPC, CBO, and/or SIZE1 values need to be prioritized against the rest, since refactoring them can yield improvements in TD Principal and Interest.

6 Implications to Practitioners and Researchers

Implications to Practitioners In terms of practitioners, based on findings of this study, we propose a prioritization approach that relies on the "*Software Guidebook and Debt Calculator*" (Eisenberg 2012). We adopt the coloring schema that is proposed by Eisenberg (Eisenberg 2012) and we use as metrics the important RIPs. Therefore, we propose the development of a 2D array: rows correspond to the RCs and as columns to the significant RIPs (MPC, CBO, and SIZE1 aggregated with the MAX function). Then a 3-step approach takes place:

1 we sort the RCs by each metric, and we color the top-10%.
2.1 assign a RED color to RCs that are colored for all metrics.
2.2 assign an ORANGE color to RCs that are colored for 2 out of 3 metrics.
2.3 assign a YELLOW color to RCs that are colored for 1 out of 3 metrics.
3 we explore the RCs whose metric scores exceed by 2-times the mean score of samples, and for those we upgrade the coloring assignment (e.g., from ORANGE to RED).

Table 7 GLMMs Estimated Parameters for Significant RIPs (TD Interest)

Aggregation	Model	CC	DIT
Sum	Estimate	-0.220	-0.231
	SE	0.099	0.089
	OR	0.8025	0.7937
	p	0.026	0.009
Max	Estimate	-0.371	-0.415
	SE	0.200	0.154
	OR	0.6900	0.6603
	p	0.046	0.007

Table 8 GLMMs Estimated Parameters for Significant RIPs (TD Principal AND TD Interest)

Aggregation	Model	MPC	CBO	SIZE1
Mean	Estimate	0.192	–	0.194
	SE	0.090	–	0.095
	OR	1.212	–	1.214
	p	0.033	–	0.042
Sum	Estimate	0.109	–	–
	SE	0.053	–	–
	OR	1.115	–	–
	p	0.040	–	–
Max	Estimate	0.162	0.194	0.147
	SE	0.073	0.094	0.074
	OR	1.176	1.214	1.158
	p	0.026	0.039	0.047

As an example, we demonstrate this process on the Apache Pinot project for a specific commit.[4] As refactoring candidates, we used five refactoring opportunities obtained through the Smell Detector Merger (Ichtsis et al. 2022) tool that validates the existence of a smell based on the intersection of multiple tools. By following the steps, we described above, we end up with the coloured RCs shown in Table 9. So, given our proposed strategy the refactoring that has the highest chance of achieving a greater impact is the ***Duplicate Code #1*** (more details about each step of this example can be found in Appendix A). We need to note that this study can not answer all the questions that might be stated in the refactoring process. For inctance: *"How many refactorings of this list MUST I apply?"*, since the answer to this question would require additional information, such as the timeframe and the budget that can be devoted to the refactoring session. However, given the available budget, the team can opt to apply refactorings, picking from the top of the prioritized list.

Implications to Researchers From this study, we can extract two types of implications to researchers: (a) from a methodological perspective; and (b) from an outcome perspective. On the one hand (*methodological implications*), through this work we have validated that treating software engineering problems as nested ones is both feasible and fitting, in the

4 https://github.com/apache/pinot/commit/7d09489c5b939666c0561b6301c9287ef34ea239

Table 9 Example for Process

RC	MPC	CBO	SIZE1
Duplicate Code #1 *BaseDistinctAggregateAggregationFunction.java* *DistinctCountSmartHLLAggregationFunction.java*	35	8	845
Duplicate Code #2 *TextContainsFilterOperator.java* *TextMatchFilterOperator.java*	8	10	37
God Class *DataBlockBuilder.java*	15	8	487
Long Method *InTransformFunction.java*	15	7	207
Duplicate Code #3 *MinAggregationFunction.java*	14	5	240

We sort the RCs by each metric, and we color the top-10%, 2.1 assign a RED color to RCs that are colored for all metrics, 2.2 assign an ORANGE color to RCs that are colored for 2 out of 3 metrics, 2.3 assign a YELLOW color to RCs that are colored for 1 out of 3 metrics. We explore the RCs whose metric scores exceed by 2-times the mean score of samples, and for those we upgrade the coloring assignment (e.g., from ORANGE to RED)

sense that an important fraction of mining software repositories studies is extracting information from multiple projects, and either report the results per project, or cumulatively for the complete population. Although such approaches are not faulty, the experimental setup of this work explicitly considers the nesting of units of analysis within different projects and does not "*hide*" the fact that different projects can be a confounding factor. In that sense, we champion the experimental setup relying on nested statistical analysis, such as MEMs. The second methodological implication of this work is related to the use of Pull Requests to extract information from grouped commits that serve a common goal. We believe that such a data collection approach can be beneficial for various study setups that currently work on the commit level, which, however, loses the context of the change that is applied during the commit. The main benefits of working with PRs instead of commits are: (a) a PR has a specific purpose / goal that can be studied by researchers, and this goals is not the subjective assessment of the research team, but a characterization of the development team based on their expertise; and (b) the fact that since a PR is a change chunk larger in size than the commit, it has the potential to be related to more meaningful and impactful changes, which however can still be treated as a unit, since they serve a common purpose.

On the other hand (*outcomes-based implications*), our study has validated related works that support that a refactoring is not always having a Positive impact on quality (Alshayeb 2009; Nikolaidis et al. 2022), confirming the motivation of investigating RIPs. The findings on the importance of specific RIPs on TD Principal and TD Interest, opens future work directions in the sense that following up on this explanatory analysis, prediction and classification models can be built, so that refactoring suggestion tools can prioritize the extracted opportunities. In this direction, we plan to further work on the current dataset to train and validate such models, and then integrate them in the Smell Detector Merger (Ichtsis et al. 2022) to equip it with prioritization functionality. Finally, we aim at an empirical validation on the usability and effectiveness of the proposed approach and tool in an industrial setting. Such a study would be more relevant if it is conducted as a human study, in which we would validate that the prioritization offered by the tool would match the "*gut feeling*" of experienced software architects and quality managers.

7 Threats to Validity

This section discusses potential threats to our study's validity, as defined in the guidelines of Runeson et al. (Runeson et al. 2012).

Construct Validity In any study, the measured phenomena might differ from the actual ones, leading to construct validity threats. For the current study involving the notions of TD Principal and Interest threats arise from the tooling employed to assess them. For measuring principal, we relied on SonarQube which is one of the most frequently used tools (Avgeriou et al. 2021; Alves et al. 2016a; Martini et al. 2018). Yli-Huumo et al. (Yli-Huumo et al. 2016) analyzed the practices of 8 development teams and identified SonarQube as the most used tool for TDM. However, despite its wide acceptance, it focuses only on code TD ignoring other manifestations of TD such as debt in requirements, architecture, build processes and tests. We should note that while SonarQube estimates can be configured by modifying the remediation time for individual TD issues, most research studies have not performed any such configuration (Schnappinger et al. 2019).

The measurement of TD Interest is far more challenging than the quantification of Principal, primarily because the assessment of Interest requires the anticipation of future modifications as well as the knowledge of the maintenance effort for an optimal version of the analyzed system (i.e. one that is debt-free). Both future maintenance activities and the notion of an ideal state of software are unknown. Therefore, TD Interest can only be assessed through proxies and by making certain assumptions. In this study, we measured TD Interest through the use of selected software metrics and by assessing the distance of any system class from its best peer. The selection of metrics was based on empirical evidence in the literature indicating that a combination of metrics can serve as a reliable maintainability predictor (Riaz et al. 2009b). The model for synthesizing the values in a unified value for TD Interest relies on solid mathematical calculations, given the assumption that maintenance effort is inversely proportional (linearly) to maintainability. This assumption, although it cannot be validated without a controlled experiment, relies on previous studies (Ampatzoglou et al. 2018; Ampatzoglou et al. 2016) and is considered as intuitive by the authors of this paper.

Furthermore, PRs labeled as 'Refactorings' have been used as a mechanism for retrieving documented refactorings in the history of a software project. We acknowledge that this approach might have missed undocumented individual refactoring applications or PRs where refactoring activities are designated using a different label. Nevertheless, labeled PRs constitute a reliable source for investigating the impact of systematic and intentional refactoring activities.

External Validity The external validity of the study may be threatened by the possibility that different projects using different programming languages or build systems may yield different observations. However, we argue that the chosen projects, due to their size and complexity, provide a realistic sample of non-trivial, real-world systems. Furthermore, while the Apache Foundation is a credible organization with diverse projects, their practices may not fully represent those of other large projects. To address this, one-third of the analyzed projects comprise non-trivial systems from outside the Apache Foundation. Lastly, it should be noted that the study's results are not applicable to non-object-oriented systems as properties such as inheritance, coupling, and cohesion, which are used to assess TD, are only applicable to OO software modules.

Reliability To mitigate potential threats to reliability, our study involved three researchers in data collection and analysis. Moreover, samples of the analysis output from different steps were manually inspected by two additional researchers for irregularities and for consistency with the proposed study design. Our results showed no irregularities, and all output from different steps was consistent with the proposed study design. Finally, we described the procedures of the data collection and analysis in as much detail as possible and the used tools are publicly available.

8 Conclusions

The impact of refactoring activities in software projects can be positive, neutral, or negative, depending on the context in which the refactoring is applied. In this study, we investigated the impact of refactoring activities on TD accumulation, focusing on the role of aggregated metrics at the 'refactoring candidate' level as predictors of the refactoring impact. Through descriptive and exploratory analytics, we found that in most cases, an improvement of TD Principal through refactoring is usually associated with an improvement of TD Interest as well. Our exploratory data analysis through visualization techniques revealed that most of the aggregated RC-level metrics can be considered as important predictors that may affect the outcome of a refactoring activity, regardless of the aggregation function for TD Principal, whereas the MAX function works better for TD Interest assessment (RQ_1). Furthermore, we identified a subset of aggregated RC-level metrics that presented a statistically significant effect on the refactoring impact on TD Principal and TD Interest. By focusing on metrics, the results suggested that RCs involving classes with excessive MPC, CBO, and/or SIZE1 values need to be prioritized against the rest, since refactoring them can yield improvements in TD Principal and Interest (RQ_2). Overall, the results of our study highlight the importance of considering aggregated RC-level metrics when evaluating the impact of refactoring activities on TD accumulation. Software developers and project managers can use these findings to make more informed decisions regarding refactoring activities and prioritize refactoring efforts based on the most relevant aggregated RC-level metrics.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/s10664-023-10412-w.

Data Availability The dataset generated and analyzed during the current study is available online.

Declarations

Conflict of Interest The authors declare that they have no conflict of interest.

References

AlOmar EA, Mkaouer MW, Ouni A, Kessentini M (2019) On the Impact of Refactoring on the Relationship between Quality Attributes and Design Metrics, 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Porto de Galinhas, Brazil, pp. 1-11

Alshayeb M (2009) Empirical investigation of refactoring effect on software quality. Inf Softw Technol 51(9):1319–1326

Alves NSR, Mendes TS, de Mendonça MG, Spínola RO, Shull F, Seaman C (2016) Identification and management of technical debt: A systematic mapping study. Inf Softw Technol 70:100–121 (**Elsevier**)

Alves NSR, Mendes TS, Mendonca MGd, Spınola RO, Shull F, Seaman C (2016) Identification and management of technical debt: A systematic mapping study. Inf Softw Technol 70:100–121

Ampatzoglou A, Ampatzoglou A, Chatzigeorgiou A, Avgeriou P (2015a) The financial aspect of managing technical debt: A systematic literature review. Inf Softw Technol. 64:52–73

Ampatzoglou A, Ampatzoglou A, Avgeriou P, Chatzigeorgiou A (2015b) "Establishing a framework for managing interest in technical debt" in 5th International Symposium on Business Modeling and Software Design (BMSD), Italy

Ampatzoglou AA, Ampatzoglou A, Avgeriou P, Chatzigeorgiou A (2016) A Financial Approach for Managing Interest in Technical Debt, 2015 International Symposium on Business Modeling and Software Design (BMSD). Springer

Ampatzoglou, Michailidis A, Sarikyriakidis C, Ampatzoglou A, Chatzigeorgiou A, Avgeriou P (2018) A framework for managing interest in technical debt: an industrial validation, in Proceedings of the 2018 International Conference on Technical Debt, pp. 115–124

Ampatzoglou A, Tsintzira AA, Arvanitou EM, Chatzigeorgiou A, Stamelos I, Moga A, Heb R, Matei O, Tsiridis N, Kehagias D (2019) Applying the Single Responsibility Principle in Industry: Modularity Benefits and Trade-offs, 23rd International Conference on the Evaluation and Assessment in Software Engineering (EASE' 19), ACM, Copenhagen, Denmark, 14–17

Ampatzoglou A, Mittas N, Tsintzira AA, Ampatzoglou A, Arvanitou EM, Chatzigeorgiou A, Avgeriou P, Angelis L (2020) Exploring the relation between technical debt principal and interest: an empirical approach. Inform Software Technol 128:106391. https://doi.org/10.1016/j.infsof.2020.106391

Arvanitou EM, Ampatzoglou A, Chatzigeorgiou A, Avgeriou P (2016) Software Metrics Fluctuation: A Prop-erty for Assisting the Metric Selection Process. Inf Softw Technol 72(4):110–124 (**Elsevier**)

Arvanitou EM, Argyriadou P, Koutsou G, Ampatzoglou A, Chatzigeorgiou A. "Quantifying TD Interest: Are we Getting Closer, or Not Even That?", 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA' 22), IEEE Computer Society, August 2020, Gran Canaria, Spain

Avgeriou P, Taibi D, Ampatzoglou A, Arcelli Fontana F, Besker T, Chatzigeorgiou A, Lenarduzzi V, Martini A, Moschou N, Pigazzini I, Saarimäki N, Sas D, Soares de Toledo S, Tsintzira A (2021) An overview and comparison of technical debt measurement tools. Journals and Magazines: IEEE Software 38(3):61–71. https://doi.org/10.1109/MS.2020.3024958

Calefato F, Lanubile F, Novielli N (2017) A Preliminary Analysis on the Effects of Propensity to Trust in Distributed Software Development, 2017 IEEE 12th International Conference on Global Software Engineering (ICGSE), Buenos Aires, Argentina, pp. 56–60

Campbell GA, Papapetrou PP (2013) SonarQube in action. Manning Publications. https://www.amazon.com/SonarQube-Action-G-Ann-Campbell/dp/1617290955

Chaparro O, Bavota G, Marcus A, Penta MD (2014) On the Impact of Refactoring Operations on Code Quality Metrics, 2014 IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, pp. 456–460

Charalampidou S, Ampatzoglou A, Chatzigeorgiou A, Gkortzis A, Avgeriou P (2016) Identifying extract method refactoring opportunities based on functional relevance. IEEE Trans Software Eng 43(10):954–974

Chatzigeorgiou, Ampatzoglou A, Ampatzoglou A, Amanatidis T (2015) Estimating the breaking point for technical debt, 2015 IEEE 7th International Workshop on Managing Technical Debt (MTD), Bremen, Germany, pp. 53-56

Digkas G, Chatzigeorgiou A, Ampatzoglou A, Avgeriou P (2022) Can Clean New Code Reduce Technical Debt Density? IEEE Computer Society, Transactions on Software Engineering

Du Bois B and Mens T (2003) Describing the impact of refactoring on internal program quality, in International Workshop on Evolution of Large-scale Industrial Software Applications, pp. 37–48

Eisenberg RJ (2012) A threshold-based approach to technical debt. SIGSOFT Softw Eng Notes 37:1–6

Falessi D, Rusfso B, Mullen K (2017) What if i had no smells? in 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, pp. 78–84

Falessi D, Russo B, Mullen K (2017) What if I Had No Smells? 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Toronto, ON, Canada, 2017, pp. 78-84

Fowler M, Beck K (1999) “Refactoring: improving the design of existing code”, ser. Addison-Wesley, In Addison Wesley object technology series

Fowler M, Beck K, Brant J, Opdyke W, Roberts D (1999) Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional, 1st Edition

Ge X, DuBose QL, Murphy-Hill E (2012) Reconciling manual and automatic refactoring," 2012 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, pp. 211–221

Gousios G, Zaidman A, Storey M-A and Deursen AV (2015) Work Practices and Challenges in Pull-Based Development: The Integrator’s Perspective, 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, Italy, pp. 358–368

Hastings T and Walcott KR (2022) Continuous Verification of Open-Source Components in a World of Weak Links, 2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Charlotte, NC, USA, pp. 201-207

Higo Y, Matsumoto Y, Kusumoto S, Inoue K (2008) Refactoring Effect Estimation Based on Complexity Metrics, 19th Australian Conference on Software Engineering (aswec 2008), Perth, WA, Australia, pp. 219–228

Ichtsis A, Mittas N, Ampatzoglou A, Chatzigeorgiou A (2022) Merging Smell Detec-tors: Evidence on the Agreement of Multiple Tools, 2022 IEEE/ACM International Conference on Technical Debt (TechDebt), Pittsburgh, PA, USA, pp. 61–65

Ivers J, Nord RL, Ozkaya I, Seifried C, Timperley CS, Kessentini M (2022) Industry experiences with large-scale refactoring, 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, USA, Nov. pp. 1544–1554

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

Kataoka Y, Imai T, Andou H, Fukaya T (2002) A quantitative evaluation of maintainability enhancement by refactoring, International Conference on Software Maintenance, pp. 576–585

Kataoka Y, Imai T, Andou H, Fukaya T (2002) A quantitative evaluation of maintainability enhancement by refactoring, International Conference on Software Maintenance, 2002. Proceedings., Montreal, QC, Canada, pp. 576–585

Kitchenham B, Pfleeger SL (1996) Software quality: the elusive target [special issues section]. IEEE Softw 13(1):12–21

Kurbatova Z, Veselov I, Golubev Y, Bryksin T (2020) June. Recommendation of move method refactoring using path-based representation of code. In Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops (pp. 315–322)

Lefever J, Cai Y, Cervantes H, Kazman R, Fang H (2021) On the lack of consensus among technical debt detection tools. International Conference on Software Engineering (SEIP); 121–130

Lehman MM, Ramil JF, Wernick PD, Perry DE, Turski WM (1997)Metrics and laws of software evolution-the nineties view. In: Proceedings fourth international software metrics symposium. IEEE, Albuquerque, NM, USA, pp 20–32. https://doi.org/10.1109/METRIC.1997.637156

Lenarduzzi V, Nikkola V, Saarimäki N, Taibi D (2021) Does code quality affect pull request acceptance? An empirical study. J Syst Softw 171:110806

Letouzey J-L (2012) The sqale method for evaluating technical debt, in 2012 Third International Workshop on Managing Technical Debt (MTD). IEEE, pp. 31–36

Li Z, Avgeriou P, Liang P (2015) A systematic mapping study on technical debt and its management. J Syst Softw 101:193–220

Martini A, Besker T, Bosch J (2018) Technical debt tracking: Current state of practice: A survey and multiple case study in 15 large organizations. Sci Comput Program 163:42–61

Mavridis A, Ampatzoglou A, Stamelos I, Sfetsos P, Deligiannis I (2012) Selecting refactorings: an option based approach. In 2012 Eighth International Conference on the Quality of Information and Communications Technology (pp. 272–277). IEEE.

Meananeatra P, Rongviriyapanish S, Apiwattanapong T (2011) Using software metrics to select refactoring for long method bad smell. 8th Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI) Association of Thailand-Conference 2011 (pp. 492–495). IEEE

Moreira Soares D, de Lima Júnior ML, Murta L, Plastino A (2021) What factors influence the lifetime of pull requests. Softw Pract Exp 51(6):1173–1193

Moser R, Abrahamsson P, Pedrycz W, Sillitti A, Succi G (2008) A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team, in Balancing Agility and Formalism in Software Engineering, Meyer B, Nawrocki JR, Walter B, Eds., in Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 252–266

Murphy-Hill E, Parnin C, Black AP (2011) How we refactor, and how we know it. IEEE Trans Software Eng 38(1):5–18

Murphy-Hill E, Black AP (2008) Breaking the barriers to successful refactoring: observations and tools for extract method. In Proceedings of the 30th international conference on Software engineering (pp. 421–430)

Nikolaidis N, Zisis D, Ampatzoglou A, Mittas N, Chatzigeorgiou A (2022) Using Machine Learning to Guide the Application of Software Refactorings: A Preliminary Exploration, 6th International Workshop on Machine Learning Techniques for Software Quality Evolution (Maltesque '22), ACM

Nikolaidis N, Ampatzoglou A, Chatzigeorgiou A, Mittas N, Konstantinidis E, Bamidis P (2023) Exploring the Effect of Various Maintenance Activities on the Accumulation of TD Principal, 6th International Conference on Technical Debt (TechDEBT' 23), Melbourne, Australia

Nikolaidis N, Mittas N, Ampatzoglou A, Arvanitou EM, Chatzigeorgiou A (2023) Assessing TD Macro-Management: A Nested Modelling Statistical Approach. Trans Softw Eng

Riaz M, Mendes E, Tempero E (2009b) "A systematic review of software maintainability prediction and metrics", 3rd International Symposium on Empirical Software Engineering and Measurement. IEEE, Florida, USA, pp 367–377

Riaz M, Mendes E, Tempero E (2009) A systematic review of software maintainability prediction and metrics, 2009 3rd International Symposium on Empirical Software Engineering and Measurement, Lake Buena Vista, FL, USA, pp. 367-377

Runeson P, Host M, Rainer A, Regnell B (2012) Case study research in software engineering: guidelines and examples. John Wiley & Sons. https://doi.org/10.1002/9781118181034

Schnappinger M, Osman MH, Pretschner A, Fietzke A (2019) Learning a classifier for prediction of maintainability based on static analysis tools Proceedings of the 27th International Conference on Program Comprehension, IEEE Press, pp. 243–248

Sharma T, Mishra P, Tiwari R (2016) "Designite: a software design quality assessment tool", 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities. NY, USA, May, New York, pp 1–4

Silva D and Valente MT (2017) Refdiff: detecting refactorings in version histories. 14th International Conference on Mining Software Repositories, pages 269–279. IEEE Press

Silva MCO, Valente MT, Terra R (2016) Does technical debt lead to the rejection of pull requests?, ArXiv Prepr. ArXiv160401450

Smiari P, Bibi S, Ampatzoglou A, Arvanitou E-M (2022) Refactoring embedded software: A study in healthcare domain. Inf Softw Technol 143:106760

Soetens QD, Demeyer S (2010) Studying the Effect of Refactorings: A Complexity Metrics Perspective, 2010 Seventh International Conference on the Quality of Information and Communications Technology, Porto, Portugal, pp. 313–318

Stroggylos K, Spinellis D (2007) "Refactoring–does it improve software quality?" in Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops. IEEE 2007:10–10

Stroulia E and Kapoor R (2001) Metrics of refactoring-based development: An experience report. In OOIS 2001, pages 113–122. Springer

Tsantalis N, Ketkar A, Dig D (2022) RefactoringMiner 2.0. IEEE Transact Software Eng 48(3):930–950. https://doi.org/10.1109/TSE.2020.3007722

Tsantalis N, Chaikalis T, Chatzigeorgiou A (2008) JDeodorant: Identification and Removal of Type-Checking Bad Smells, 2008 12th European Conference on Software Maintenance and Reengineering, Athens, Greece, pp. 329–331

Tsintzira A-A, Ampatzoglou A, Matei O, Ampatzoglou A, Chatzigeorgiou A, Heb R (2019) Technical debt quantification through metrics: an industrial validation, in 15th China-Europe International Symposium on software engineering education

Van Koten C, Gray AR (2006) An Application of Bayesian Network for Predicting Object-Oriented Software Maintainability. Inform Software Tech 48(1):59–67

Wilking D, Kahn UF, Kowalewski S (2007) An empirical evaluation of refactoring. e-Informatica 1(1):27–42

Yli-Huumo J, Maglyas A, Smolander K (2016) How do software development teams manage technical debt?–An empirical study. J Syst Softw 120:195–218 (**Elsevier**)

Zhang X, Yu Y, Gousios G, Rastogi A (2023) Pull request decisions explained: an empirical overview. IEEE Transact Software Eng 49(2):849–871. https://doi.org/10.1109/TSE.2022.3165056

Zhou Y, Xu B (2008) Predicting the maintainability of open-source software using design metrics. Wuhan Univ J Nat Sci 13(1):14–20

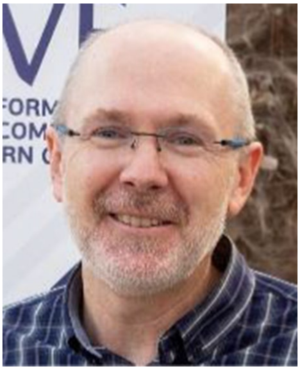
Zou W, Xuan J, Xie X, Chen Z, Xu B (2019) How does code style inconsistency affect pull request integration? an exploratory study on 117 github projects, Empir Softw Eng. 24(6)

Nikolaos Nikolaidis is an PhD student at the Department of Applied Informatics in the University of Macedonia, Greece. He received a BSc in Applied Informatics from the same university in 2018. He is currently employed as a research associate in the Software Engineering group of University of Macedonia, working on multiple research projects. His research interests are technical debt management, mining software repositories, and software quality assurance.

Dr. Nikolaos Mittas Dr. Nikolaos Mittas is an Associate Professor in the Department of Chemistry at the International Hellenic University (IHU), Greece. He received a BSc degree in Mathematics from the University of Crete and MSc and PhD degrees in Informatics from the Aristotle University of Thessaloniki (AUTH), Greece. His current research interests are focused on the application of statistics and data analytics in the area of software engineering, project management, and competence management.

Dr. Apostolos Ampatzoglou is an Associate Professor in the Department of Applied Informatics in University of Macedonia (Greece), where he carries out research and teaching in the area of software engineering. Before joining University of Macedonia, he was an Assistant Professor in the University of Groningen (Netherlands). He holds a BSc on Information Systems (2003), an MSc on Computer Systems (2005) and a PhD in Software Engineering by the Aristotle University of Thessaloniki (2012). He has published more than 100 articles in international journals and conferences, and is/was involved in over 15 R&D ICT projects, with funding from national and international organizations. His current research interests are focused on technical debt management, software maintainability, reverse engineering software quality management, open-source software, and software design.

Dr. Daniel Feitosa is an Assistant Professor in the University of Groningen (Netherlands). He holds a B.S. of Computer Science in the Institute of Mathematics and Computational Sciences of the University of São Paulo, with specialization in Software Engineering. He has completed his M.Sc. in the Institute of Mathematics and Computational Sciences of the University of São Paulo, within the Software Engineering group (LabES) and Mobile Robotics group (LRM). Finally, Ph.D. in the Department of Computer Science of the University of Groningen, within the Software Engineering and Architecture group (SEARCH). Doctoral dissertation: Applying Patterns in Embedded Systems Design for Managing Quality Attributes and their Trade-offs. He has published numerous articles in international journals and conferences, and is/was involved in various research projects, with funding from national and international organizations. His current research interests are focused on technical debt management, software maintainability, reverse engineering software quality management, open-source software, and software design.



Dr. Alexander Chatzigeorgiou is a Professor of Software Engineering in the Department of Applied Informatics and Vice Rector of Extroversion and International Relations at the University of Macedonia, Thessaloniki, Greece. He received the Diploma in Electrical Engineering and the PhD degree in Computer Science from the Aristotle University of Thessaloniki, Greece, in 1996 and 2000, respectively. From 1997 to 1999 he was with Intracom S.A., Greece, as a software designer. His research interests include object-oriented design, software maintenance, technical debt and evolution analysis. He has published more than 150 articles in international journals and conferences and participated in a number of European and national research programs. He is a Senior Associate Editor of the Journal of Systems and Software and an Associate Editor of the ACM Transactions on Software Engineering and Methodology.