Using Machine Learning to Guide the Application of Software Refactorings: A Preliminary Exploration

Nikolaos Nikolaidis
Department of Applied Informatics
University of Macedonia
Thessaloniki, Greece
nnikolaidis@uom.edu.gr

Dimitrios Zisis
Accenture
Thessaloniki, Greece
zisisndimitris@gmail.com

Apostolos Ampatzoglou
Department of Applied Informatics
University of Macedonia
Thessaloniki, Greece
a.ampatzoglou@uom.edu.gr

Nikolaos Mittas
Department of Chemistry
International Hellenic University
Kavala, Greece
nmittas@chem.ihu.gr

Alexander Chatzigeorgiou
Department of Applied Informatics
University of Macedonia
Thessaloniki, Greece
achat@uom.edu.gr

ABSTRACT

Refactorings constitute the most direct and comprehensible approach for addressing software quality issues, stemming directly from identified code smells. Nevertheless, despite their popularity in both the research and industrial communities: (a) the effect of a refactoring is not guaranteed to be successful; and (b) the plethora of available refactoring opportunities does not allow their comprehensive application. Thus, there is a need of guidance, on when to apply a refactoring opportunity, and when the development team shall postpone it. The notion of interest, forms one of the major pillars of the Technical Debt metaphor expressing the additional maintenance effort that will be required because of the accumulated debt. To assess the benefits of refactorings and guide when a refactoring should take place, we first present the results of an empirical study assessing and quantifying the impact of various refactorings on Technical Debt Interest (building a real-world training set) and use machine learning approaches for guiding the application of future refactorings. To estimate interest, we rely on the FITTED framework, which for each object-oriented class assesses its distance from the best-quality peer; whereas the refactorings that are applied throughout the history of a software project are extracted with the RefactoringMiner tool. The dataset of this study involves 4,166 refactorings applied accriss 26,058 revisions of 10 Apache projects. The results suggest that the majority of refactorings reduce Technical Debt interest; however, considering all refactoring applications, it cannot be claimed that the mean impact differs from zero, confirming the results of previous studies highlighting mixed effects from the application of refactorings. To alleviate this problem, we have built an adequately accurate (~70%) model for the prediction of whether or not a refactoring should take place, in order to reduce Technical Debt interest.

CCS CONCEPTS

• Software and its engineering→Software creation and management→Software post-development issues→Maintaining software

KEYWORDS

Technical Debt, Refactoring, Interest, Software Quality

ACM Reference format:

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

1 Introduction

The Technical Debt (TD) metaphor captures the amount of effort and the associated cost that a development team "borrows", by opting for a "quicker" but "non-optimal" approach in terms of software quality—implying that interest will have to be paid. Technical Debt interest expresses the additional effort that will be spent during later software maintenance tasks, exactly because inefficiencies are present. Interest is of great concern to software development teams as it essentially describes the future cost of '*sweeping problems under the carpet*'. Empirical studies have shown that code TD usually increases as systems grow, but TD density (TD normalized over the total lines of code) may decrease for some software projects [1]. In other words, TD can be repaid and under circumstances it can be reduced.

Generally, there are two approaches for Technical Debt repayment: The most widely adopted strategy is through the application of refactorings to purposefully eliminate code, design or architectural smells and implementation flaws that may exist. An alternative approach can be followed by adopting Quality Gates that ensure the introduction of "cleaner" new code, i.e., new code that has limited or zero TD issues [2]. While several empirical studies have investigated the impact of refactoring application on various aspects of software quality, the results point to mixed conclusions (see Section 2). Thus, an important question is what are the appropriate cases of such refactorings opportunities to be applied, so as to ensure a positive impact. In related studies, we have identified none that focuses on TD interest.

In this study, we estimate the amount of interest per file (relying on the FITTED framework [10]), and subsequently we investigate the sign and extent of refactorings effect on TD interest. The first goal of our study is to assess whether refactorings are an effective way of preventing the increase of TD interest and to investigate which of the refactorings have a positive or negative impact on interest. While the second goal is to leverage the dataset that will be created, in order to achieve the first goal, to create a model that can predict whether a refactoring should take place to positively affect interest. To this end, we analyze 26,058 commits extracted from 10 open-source projects looking for refactoring applications through the RefactoringMiner tool. Our analysis is facilitated by a tool that has been developed to identify the files in each commit which underwent a pure refactoring (i.e. without any associated maintenance other than refactoring) and calculate the change in the interest of that file for the pre- and refactoring-commits. Thus, we obtained results for 4,166 refactoring applications enabling us to study the average impact of refactoring on interest, but also the impact per refactoring type. Finally, we used the previous dataset to create a predictive model by using the random forest learning method, which enables us to guide practitioners on when they should apply a refactoring and when postpone it.

The rest of the paper is organized as follows: In Section 2 we discuss related, while in Section 3 we briefly outline how interest is calculated. 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. We identify threats to the validity of the study in Section 6 and finally, we conclude in Section 7.

2 Related Work

In this section, we discuss previous studies that investigate the impact of code refactorings on various aspects of software quality. Murphy-Hill et al. [3] investigated the habits of developers in terms of refactorings. They found that developers rarely perform refactoring related activities. Stroggylos and Spinellis [4] inspected the logs in the version control systems of four open-source software projects to extract the revisions where software refactorings had taken place. The findings reveal that, despite the expectation that the refactorings improve the quality of the software, the measurements in the examined systems show the opposite. In particular, it was observed that the code refactorings caused a slight increase in cohesion and coupling related metrics.

Kataoka et al. [5] evaluated the impact of the "*Extract Method"* and the "*Extract Class"* refactorings on a software project's maintainability, written in C++, using coupling metrics. The results indicate that refactorings magnify system maintainability. Bois and Mens [6] proposed formalism based on abstract syntax tree representation of the source-code, extended with cross-references to describe the impact of refactoring on internal program quality. They focused on three refactoring methods: "*Encapsulate Field*", ''Pull up Method", and ''Extract Method". In another study, Alshayeb [7] concluded that the application of refactorings does not necessarily improve external quality characteristics, such as adaptability, maintainability and comprehensibility. By applying refactoring techniques, as defined by Fowler, to three software systems and measuring the effect on selected software metrics, a vast discrepancy in the effect of the refactorings was revealed. The author concluded that it was not possible to corroborate that software refactorings as a general practice can improve quality.

Stroulia and Kapoor [8] investigated the effect on size and coupling measures after the application of refactoring. The results in Stroulia and Kapoor's work show that size and coupling metrics decreased after refactorings. Also noteworthy is the study by Wilking et al. [9], who conducted a controlled experiment to investigate how refactorings affect the conservation and modification of projects. The results of their experiment proclaim that there is no direct effect of software refactoring leading to improved maintainability.
The majority of the findings of the above studies agree on the limited practical adoption of software refactorings and on a rather mixed effect on the quality of a project, at least on quality aspects that can be quantified.

3 Technical Debt Interest

To measure code Technical Debt interest, we adopt the FITTED approach [10]. According to the perspective of that methodology, each artifact but also the system as a whole is represented by two concepts: the actual artifact and a hypothetical optimum. Being at the optimal level, it takes less effort to maintain or extend the code. In contrast, at the level of the actual system, relatively more effort is required for maintenance and extension. The difference in effort is defined as the Technical Debt Interest—see Figure 1.

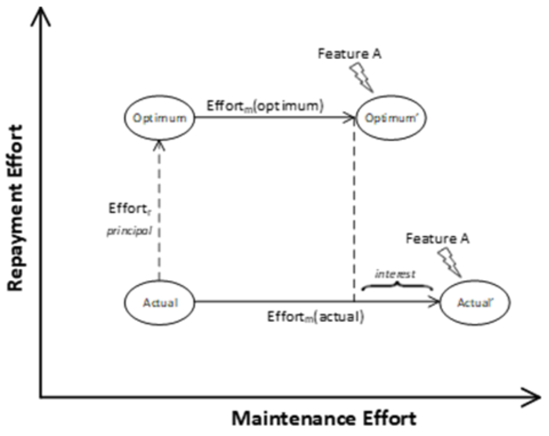

Figure 1: FITTED High-Level Rationalle

The proposed interest measurement approach is based on historic 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 procedure followed to find the optimum of a class is as follows: (a) Find the 5 closest neighbors (classes of the system) of the class under study, based on quality characteristics, such as complexity, coupling, size, coherence and inheritance, (b) Having these 5 neighboring classes, an aggregate function (minimum) is applied for each metric of the aforementioned quality characteristics. Thus, a theoretical optimum peer is obtained for each examined class.

The ratio of the quality of the class under study over the quality of its optimum peer determines the additional maintenance effort for that class, by projecting past maintenance effort. Despite its assumptions, the FITTED methodology extracts an 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. [14] the FITTED-based interest assessment 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.

4 Case Study Design

In this section, we present the design of the case study, reported based on the linear-analytic structure [11].

4.1 Research Questions

We study the impact of refactorings on TD interest through three research questions, formulated as follows:

RQ_1: What is the average impact of refactoring application on TD Interest?

RQ_2: What is the impact of each refactoring type on TD Interest?

RQ_3: Can we predict whether or not a refactoring should take place, based on its foreseen impat on TD interest?

The answer to the **RQ_1** we will unveil whether refactorings have a positive or negative effect on the TD interest of the involved files. A negative effect implies that the metrics which are being used to assess the distance of the examined class / file deteriorate placing the class further away from the corresponding optimal peer. While refactorings are known to remove the targeted code smell this often comes at the cost of side effects, such as increase in the number of lines of code, methods or classes. For the **RQ_2** we acknowledges that refactoring types are quite different in nature and thus might have diverse impact on the metrics by which TD interest is assessed. Excluding refactorings which by definition are expected to have no impact on the measured interest, it becomes interesting to classify refactorings based on their positive or negative impact. Of course, one should by no means consider a refactoring exhibiting a negative impact on interest as a non-meaningful refactoring. It is known that refactorings improve code also by making it more readable and reusable, qualities which are not necessarily captured by the employed set of metrics. Finally, regarding **RQ_3**, by acknowledging the fact that the application of refactorings sometimes comes with trade-offs that deteriorate a lot the quality of the system, we will try to create a model that can shed some light on this decision-making. In other words, given the current state of the refactoring-candidate classes, we predict if the refactoring should be applied, so as to positively affect (reduce) the Technical Debt interest.

4.2 Cases and Units of Analysis

This study is characterized as a multiple, embedded case study [11], in which the cases are open-source software (OSS) projects, while the units of analysis are the files affected by refactoring in individual source code commits (per project). To retrieve data from high-quality projects that evolve over a considerable period of time and have high chances of being the subject of systematic maintenance including refactoring applications, we looked into Apache projects and investigated the projects presented in Table 1. The selection of projects was based on the following criteria:

- ***Written in Java and use Maven***. This ensures that the project can be built and can be analyzed by the RefactoringMiner to retrieve historically applied refactorings to answer RQ_1 and RQ_2 and build a training / testing set for RQ_3.
- ***Currently under development*** and thus still maintained. This criterion aims at ensuring that the projects included in the analysis are still undergoing development and therefore the studied practices will not be outdated; increasing the chances for identifying refactorings.
- ***More than 600 commits***. We have included this criterion for similar reasons to the previous one and to be able to observe longer periods throughout the history of a project, since refactoring sessions might not be part of all maintenance periods.

Table 1: Selected Projects

Project	# Commits	LoC	#Classes
Commons-IO	3492	36950	440
Commons-Lang	6576	94355	772
Commons-RDF	1303	5990	184
Flume	1832	110747	1465
Giraph	1387	38883	1359
Griffin	638	29544	144
Johnzon	840	5311	601
Maven-Archetype	1266	21357	160
OpenWebBeans	4016	7625	1330
Unomi	2402	28781	748

4.3 Data Collection

To gather the appropriate data for our study, we devised a collection plan, which is divided into 2 distinct phases as described below. Data collection relies on two well-known tools: Refactoring Minner and FITTED Interest Calculator.

Phase 1: By adopting the FITTED methodology, we measure the interest of 10 active Apache projects. The measurements concern the whole history of the master's branch commits, reflecting the production-ready state of the projects. To isolate the true change in the TD interest of each file due to a restructuring (that is, by not accounting changes in neighboring classes), we slightly change the procedure followed in FITTED: Whenever a file containing a refactoring is detected, we do not re-calculate its 'new' nearest neighbors, but retain as neighborhood the state of the classes in the pre-refactoring commit.

Phase 2: We proceed to the mapping of the changes of the files per version to the identified refactorings. For this purpose, we use the state-of-the-art tool "RefactoringMiner" [12], [13] which can detect refactorings applied in the history of a Java project. We use its API as part of our own tool in order to receive information about the kind of software refactorings that were applied per commit, the files and classes that were involved, and the exact code ranges that were affected in each of these files due to refactoring application. Apparently, it cannot be ruled out that some chunks of code that are not related to refactorings in these files may be present and, therefore, may also affect the interest of each file. That being the case, we introduce the notion of a *pure refactored* file, a concept that refers to files whose set of changes is mapped to specific refactoring implementations and only. Thus, mixed files, that is, files containing refactorings and other new or modified code as well, are excluded from our dataset.

We note that from the dataset, we have excluded certain types of refactorings, such as "*Move Class*", "*Move & Rename Class*", "*Extract Attribute*", "*Modify Variable Annotation*", "*Parameterize Variable*", "*Remove Attribute Annotation*" and "*Remove Parameter*", whose contribution to quality metrics related to complexity, size, inheritance or cohesion and, thus, to TD interest is expected to be zero. This fact has also been evaluated experimentally, since the specific types of refactorings had indeed zero effect on TD interest. For further reading, the complete dataset is available online[1].

4.4 Data Analysis

We relied on descriptive and inferential statistics for the analysis of data for answering the first 2 RQs, as follows: The distribution of positive and negative values for the calculated impact on the TD interest of each file affected by a refactoring is visualized with the use of violin plots. Violin plots illustrate numeric data distributions for one or more groups using density curves. The width of each density curve is proportional to the frequency of data points in each region. We used a violin plot for RQ_1, illustrating the distribution of refactoring impact for all observed refactorings as explained in subsection 4.3. We also report the percentage of cases in which a positive, zero, or negative impact was observed (for all refactorings and for each refactoring type). To test whether the mean of the examined population (refactoring impact for all units of analysis and impact per refactoring type) is statistically different from zero, we relied on the one-sample t-Test. The dataset meets the requirements for applying this parametric test as: (a) test variables are continuous; (b) scores on the test variable are independent; and (c) distributions are normal.

4.5 Model Building

For the creation and testing of the model, we used the RapidMiner software. As shown in Figure 2, our dataset underwent some changes before we could crate and train our model. First, we had to filter the non-clean refactoring (as we already mentioned), and create the "Interest Effect" attribute because in our dataset is present only the Interest Change in Hours or Monetary terms. The "Interest Effect" can be "*neutral*", "*positive*", or "*negative*" depending on the Interest Change being equal, less, or greater than zero respectively. Moreover, we selected the "Interest Effect" as the dependent variable, and the rest of the attributes are: (a) Cyclomatic Complexity—CC of the previous revision; (b) ines of Code—LOC of the previous revision; (c) Coupling Between Objects—CBO of the previous revision; (d) Lack of Cohesion of Methods—LCOM of the previous revision; (e) Interest of the previous revision; (f) Refactoring type; and (g) Revision count. The Cross Validation is configured to run 10 folds and the predictive model that was used is the Random Forest. The selection of quality properties relies on the most important attributes for assessing software maintainability [15][16]—inherently related to TD interest.

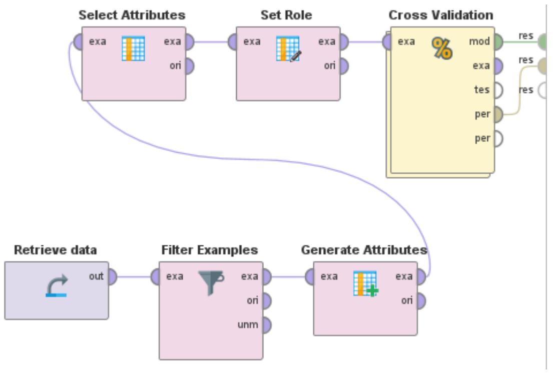

Figure 2: RapidMiner Process

5 Results and Discussion

Previous studies on the effect of refactoring application on software quality reached inconclusive results, since the impact varies among studies and among refactorings. This finding is confirmed by the present study from the perspective of TD interest: the one-sample t-test for the entire set of units of analysis (i.e. files that underwent a refactoring) yielded

[1] https://docs.google.com/spreadsheets/d/1PDmWqPts5wB_yaM0GX8ZxKOeb-dxPKdex/

results which are not statistically significant. In other words, we cannot claim that the mean of our population is statistically different from the zero value. The same observation applies to almost all of the refactoring types if the units of analysis are examined separately.

The violin plot of Fig. 2 illustrates the distribution for the percentage change of TD interest after the application of a refactoring, considering all refactorings across all examined projects. The median and the largest portion of observations is zero. To shed light into the ratio of refactored files in which a positive (reduction of TD interest), zero and negative (increase in TD interest) impact has been observed, we list the corresponding percentages in Table 2 for the entire set of observations and in Table 3 for the files affected by each refactoring. For each subset of observations we denote the corresponding number of units of analysis (N), which also highlights the popularity of various refactorings in the examined projects. In almost all cases, zero impact holds the lion share explaining the inability to reach conclusive results from the one-sample t-test. Without considering the results as statistically significant we note however that the cases where a refactoring had a positive impact are substantially more than the cases with a negative impact.

The straightforward implication from the examination of our findings is that further research is needed so as to establish evidence on the usefulness of refactorings in general and the benefit (or harm) from the application of individual refactorings. It is known that refactorings incur a trade-off: the improvement of maintainability usually comes at the cost of an increase in the number of lines of code, methods or classes and often an increase in the coupling among classes. Such side-effects can negatively affect interest thereby limiting or even cancelling any benefit from the improvement of other qualities such as complexity or cohesion.

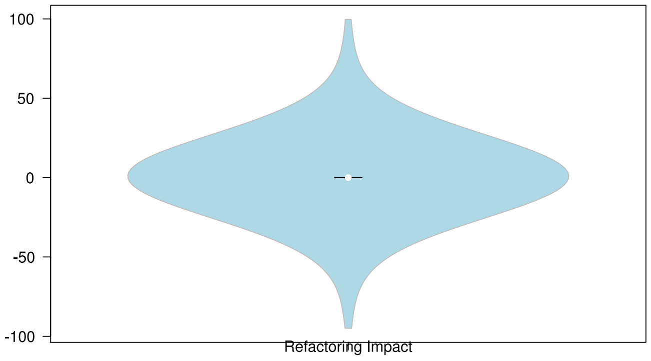

Figure 2: Distribution of percentage change of TD interest for all refactorings (Refactoring Impact)

Table 2: Percentage of Cases with Positive/Zero/Negative Impact on Interest

	Positive Impact (%)	Zero Impact (%)	Negative Impact (%)
Refactorings (N=4166)	27.58	56.72	15.70

Table 3: Percentage of Cases with Positive/Zero/Negative Impact per Refactoring (refactorings with N>10)

Add Attribute Annotation (N=56)		Add Attribute Modifier (N=111)		Add Class Annotation (N=297)	
Positive	12.50	Positive	40.54	Positive	18.52
Zero	75.00	Zero	45.05	Zero	73.40
Negative	12.50	Negative	14.41	Negative	8.08
Add Method Annotation (N=394)		Add Variable Modifier(N=175)		Add Parameter Modifier (N=93)	
Positive	19.80	Positive	52.00	Positive	46.24
Zero	36.80	Zero	29.14	Zero	40.86
Negative	43.40	Negative	18.86	Negative	12.90
Add Parameter (N=88)		Add Variable Annotation (N=14)		Change Variable Type (N=238)	
Positive	34.09	Positive	7.14	Positive	28.99
Zero	54.55	Zero	42.86	Zero	63.03
Negative	11.36	Negative	50.00	Negative	7.98
Change Attribute Access Modifier (N=63)		Change Attribute Type (N=69)		Change Parameter Type (N=173)	
Positive	38.10	Positive	23.19	Positive	26.01
Zero	52.38	Zero	73.91	Zero	68.79
Negative	9.52	Negative	2.90	Negative	5.20
Change Return Type (N=394)		Change Type Declaration Kind (N=13)		Extract Method (N=638)	
Positive	19.80	Positive	7.69	Positive	22.41
Zero	36.80	Zero	92.31	Zero	61.76
Negative	43.40	Negative	0.00	Negative	15.83
Extract And Move Method (N=330)		Extract Class (N=111)		Extract Interface (N=88)	
Positive	32.42	Positive	26.73	Positive	28.41
Zero	51.82	Zero	53.15	Zero	50.00
Negative	15.76	Negative	20.72	Negative	21.59
Extract Superclass (N=128)		Extract Subclass (N=16)		Inline Method (N=21)	
Positive	15.50	Positive	12.50	Positive	9.52
Zero	71.09	Zero	87.50	Zero	80.95
Negative	16.41	Negative	0.00	Negative	9.52
Inline Variable (N=16)		Modify Class Annotation (N=143)		Modify Method Annotation Impact (N=23)	
Positive	18.75	Positive	50.35	Positive	13.04
Zero	43.75	Zero	37.06	Zero	86.96
Negative	37.50	Negative	12.59	Negative	0.00
Move Attribute (N=11)		Move And Inline Method (N=17)		Move And Rename Method (N=12)	
Positive	27.27	Positive	35.29	Positive	8.33
Zero	54.55	Zero	35.29	Zero	75.00
Negative	18.18	Negative	29.41	Negative	16.67
Move Method (N=56)		Remove Attribute Modifier (N=48)		Remove Class Annotation (N=184)	
Positive	26.79	Positive	31.25	Positive	27.72
Zero	64.29	Zero	60.42	Zero	64.67
Negative	8.93	Negative	8.33	Negative	18.18
Remove Method Annotation (N=55)		Remove Variable Modifier (N=21)		Rename Class (N=52)	
Positive	21.82	Positive	61.90	Positive	67.31
Zero	36.36	Zero	23.81	Zero	19.23
Negative	41.82	Negative	14.29	Negative	13.46
Rename Method (N=171)		Rename Parameter (N=27)		Replace Anonymous with Lambda (N=16)	
Positive	26.32	Positive	18.52	Positive	31.25
Zero	65.50	Zero	70.37	Zero	43.75
Negative	8.19	Negative	11.11	Negative	25.00

Regarding the RQ_3, in Table 3 we present the performance of two models by using different maximal depth values for the created trees. It is clear that we could use a model to predict whether or not a refactoring should be applied, based on the effect that this change could have on the Technical Debt interest. With respect to practitioners, the application of refactorings is usually driven by the need to remove a bothersome code smell which hinders a maintenance task. Nevertheless, development teams could form a refactoring log, recording the impact of applied refactorings. Systematic evidence from past refactorings for a particular project or domain can guide future maintenance activities by taking more informed decisions. Along with the use of the predictive model the practitioners will be able to have an understanding of the impact that a specific refactoring might have beforehand. Refactorings that systematically deteriorate code quality can be discouraged by policies or automated linters.

Table 3: Accuracy and kappa metric for predictive models

	Accuracy (%)	kappa
Maximal Depth = 10	65.85	0.1
Maximal Depth = 15	69.75	0.45

6 Threats to Validity

The present study was conducted on 10 open-source projects. As a result of the relatively small sample the findings cannot be generalized to the entire population of open-source systems nor to industrial projects where more systematic or different refactoring policies might be followed. While also being too small of a dataset to have a good and not overfitting predictive model. We plan to mitigate this threat to external validity by conducting a broader study on a larger set of projects, including both open-source and industrial ones. The external validity is affected also by the fact that the target programming language is Java, because RefactoringMiner operates on Java source code.

The units of the presented study are individual files at specific commits. Thus, the granularity of the analysis is limited to files which underwent a refactoring application. In other words, this study does not consider cases where a refactoring was applied in a file along with other, non-refactoring-related maintenance. Although we do not anticipate that this parameter might have affected the results to a significant extent, it poses a threat to the construct validity of the study. However, figuring out distinct parts in the same file which are associated with refactorings or other maintenance tasks is a challenging analysis.

7 Conclusions

Refactoring application aims at improving software maintainability by removing identified code smells. In this study we have tested this axiom by investigating the impact of refactorings on TD interest, and the ability to create a predictive model to help the developers decide whether a refactoring should take place. More specifically, we identified files that underwent only refactoring activities in revisions of 10 open-source projects. By quantifying interest through a methodology that assesses the distance of an examined class from its closest optimum peer class, we determined the impact of refactorings on average and the impact of individual refactoring types. The results were inconclusive confirming previous studies about the mixed effect of refactoring applications. Nevertheless, the majority of refactorings with a non-zero effect had a positive impact on TD interest, partially validating the potential of refactorings in repaying TD. As for the predictive model, even though the dataset could be considered small, the final models are promising as they achieve a quite high accuracy.

ACKNOWLEDGMENTS

This work is partially funded by the European Union's Horizon 2020 Research and Innovation Programme through SmartCLIDE project under Grant Agreement No. 871177.

REFERENCES

[1] G. Digkas, M. Lungu, A. Chatzigeorgiou, and P. Avgeriou, "The evolution of technical debt in the apache ecosystem," in European Conference on Software Architecture. Springer, 2017, pp. 51–66.

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

[3] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," IEEE Transactions on Software Engineering, vol. 38, no. 1, pp. 5–18, 2011.

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

[5] Y. Kataoka, T. Imai, H. Andou and T. Fukaya, "A quantitative evaluation of maintainability enhancement by refactoring," International Conference on Software Maintenance, 2002. Proceedings., 2002, pp. 576-585, doi: 10.1109/ICSM.2002.1167822.

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

[7] M. Alshayeb, "Empirical investigation of refactoring effect on software quality", Information and Software Technology, Volume 51, Issue 9, 2009, Pages 1319-1326, ISSN 0950-5849, doi: 10.1016/j.infsof.2009.04.002.

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

[9] D. Wilking, U. F. Kahn, and S. Kowalewski, "An empirical evaluation of refactoring." e-Informatica, vol. 1, no. 1, pp. 27–42, 2007.

[10] A. Ampatzoglou, A. Michailidis, C. Sarikyriakidis, A. Ampatzoglou, A. Chatzigeorgiou and P. Avgeriou, "A Framework for Managing Interest in Technical Debt: An Industrial Validation," 2018 IEEE/ACM International Conference on Technical Debt (TechDebt), 2018, pp. 115-124.

[11] P. Runeson, M. Host, A. Rainer, and B. Regnell, Case Study Research in Software Engineering: Guidelines and Examples. John Wiley & Sons, 2012.

[12] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinanian and D. Dig, "Accurate and Efficient Refactoring Detection in Commit History," 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), 2018, pp. 483-494, doi: 10.1145/3180155.3180206.

[13] N. Tsantalis, A. Ketkar and D. Dig, "RefactoringMiner 2.0," in IEEE Transactions on Software Engineering, doi: 10.1109/TSE.2020.3007722.

[14] Tsintzira, A. A., Ampatzoglou, Ar., Matei, O., Ampatzoglou, Ap., Chatzigeorgiou, A., and Heb, R., "Technical Debt Quantification through Metrics: An Industrial Validation", 15th China-Europe International Symposium on Software Engineering Education (CEISEE' 19), IEEE TEMS, Lisbon-Caparica, Portugal, May 2019.

[15] M. Riaz, E. Mendes and E. Tempero, "A systematic review of software maintainability prediction and metrics," 2009 3rd International Symposium on Empirical Software Engineering and Measurement, 2009, pp. 367-377, doi: 10.1109/ESEM.2009.5314233.

[16] C. Van Koten and A.R. Gray, "An Application of Bayesian Network for Predicting Object-Oriented Software Maintainability", Inform Software Tech, 48, 1 (Jan. 2006), pp. 59 – 67.