University of Groningen

Reducing Technical Debt Density: Refactoring vs. Writing Clean New Code

Digkas, George

DOI:
[10.33612/diss.181245089](https://doi.org/10.33612/diss.181245089)

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

Publication date:
2021

[Link to publication in University of Groningen/UMCG research database](#)

Citation for published version (APA):
Digkas, G. (2021). *Reducing Technical Debt Density: Refactoring vs. Writing Clean New Code*. [Thesis fully internal (DIV), University of Groningen]. University of Groningen. <https://doi.org/10.33612/diss.181245089>

Copyright

Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

The publication may also be distributed here under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license. More information can be found on the University of Groningen website: <https://www.rug.nl/library/open-access/self-archiving-pure/taverne-amendment>.

Take-down policy

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): <http://www.rug.nl/research/portal>. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

Reducing Technical Debt Density: Refactoring vs. Writing Clean New Code

Georgios Digkas

RIJKSUNIVERSITEIT GRONINGEN

**Reducing Technical Debt Density: Refactoring vs. Writing Clean New
Code**

Proefschrift

ter verkrijging van de graad van doctor aan de
Rijksuniversiteit Groningen
op gezag van de
rector magnificus prof. dr. C. Wijmenga
en volgens besluit van het College voor Promoties.
De openbare verdediging zal plaatsvinden op
vrijdag 8 oktober 2021 om 14.30 uur

door

Georgios Digkas

geboren op 20 februari 1992
te Veroia, Griekenland

Promotores: Prof. dr. ir. P. Avgeriou

Co-promotores: Prof. dr. A.N. Chatzigeorgiou
Dr. A. Ampatzoglou

Beoordelingscommissie: Prof. dr. T. van der Storm
Prof. dr. C. Izurieta
Prof. dr. A. Martini

Abstract

Technical Debt (TD) is a powerful metaphor that represents shortcuts taken in a software development project, usually to meet business goals such as limited time or budget. Technical Debt concerns mostly invisible parts of the system (i.e. no visible features or defects) and has a detrimental effect on the maintainability and evolvability of software. Technical Debt manifests in both design and implementation constructs such as sub-optimal design decisions, design and code smells, use of impertinent technologies, lack of documentation etc.

Technical Debt cannot realistically be eliminated but it does need to be proactively *managed* in order to be kept at a sustainable level. Managing Technical Debt is comprised of a number of activities. In this thesis, we focus on two of these activities, which are reported in the literature as the most important: repayment and prevention. As its name suggests, *repayment* refers to the remediation of Technical Debt by reworking the aforementioned design and implementation constructs; consider for example refactoring the source code, optimizing a design decision or documenting architecture knowledge that has remained tacit. Similarly, *prevention* of Technical Debt refers to ensuring the underlying issues will not occur in the first place; consider for example, checking for source code violations, supporting design decision making to avoid future maintenance overhead or simply allocating enough resources to follow the right approach instead of the ‘quick and dirty’ one. **In this thesis, we investigate to what extent repayment (through refactoring) and prevention (through writing clean new code) are effective in managing Technical Debt.**

In order to address the stated problem, we first need to understand better the evolution of technical debt in practice. Specifically we need to investigate whether technical debt grows or remains stable over time, both at the macroscopic level, and at the level of individual issues and their types. Through a large-scale case study, we learned that in the majority of the systems that we studied, there is a significant increase trend on the size, number of issues, and on the complexity metrics of the project. On the other hand, the amount of technical

debt per line of code (the density of Technical Debt) decreases as the project evolves. The improvement of $TD_{density}$ is an important finding, in the sense that it contradicts the 7th of Lehman’s laws of evolution which states that the quality of software deteriorates over time. This begs the question ‘why’: is it mostly due to repayment or prevention? Specifically, in the next parts of this thesis, we investigate whether $TD_{density}$ can be decreased by refactoring pieces of code suffering from design or code inefficiencies (repayment) or by striving to introduce new code that has zero or very low technical debt (prevention).

We begin by looking into the case of refactoring. Specifically, we want to find out whether refactoring is systematically applied in practice, or whether it is more wishful thinking. Recent literature has indicated that refactoring is rather limited in practice, as developers are mostly busy with introducing new features or fixing bugs. Through a second large case study on open-source software, we found that a large percentage of technical debt is paid back during evolution; however, for the majority of the projects, the fixing rate is below 30%. This entails that refactoring is not as effective in addressing technical debt as more than 70% of the technical debt items are not fixed.

After studying the case of refactoring, we move on to the second case: does introducing new, clean code do a better job with reducing normalized technical debt? To this end, we conducted a third case study, which showed that the newly inserted code exhibits higher quality (in terms of $TD_{density}$), compared to the quality of the systems in which the new code is introduced and this was consistent in all examined projects. So it is possible that the system’s overall quality can be improved if the $TD_{density}$ of newly inserted code is systematically kept below the system’s average, for example through the use of Quality Gates.

Having established that the density of technical debt can be decreased using new, clean code, we investigate further the impact of clean code on technical debt. Specifically, we study the impact of new code on the two core concepts of technical debt: *principal* and *interest*. The principal refers to shortcomings like code smells, violations of coding rules, design smells etc. that need fixing. In contrast, technical debt interest refers to the additional maintenance effort and associated costs that stem from the very existence of technical debt items in a system.

Starting from technical debt principal, we study how it fluctuates along evolution, since a software system grows mostly through the addition of new code. Through our fourth case study, we found that the introduction of code Technical Debt Issues due to the insertion of new code is independent of time, for more than half of the projects. While this can be interpreted as an indication of project maturity, even for these stable projects, the absence of fluctuations does not necessarily imply the absence of any trend. Moreover, the number of commits made in a time period as well as the experience of developers, are not correlated to the number of introduced code Technical Debt Issues into the system (for the majority of the cases). This is counter-intuitive, but we conjecture that other confounding factors may be more relevant, such as: (a) the maturity of the project; (b) the developers’ habits; or (c)

the specific type of tasks performed in each time period.

Moving on to the second constituent of technical debt, i.e. the interest, in the final part of this thesis, we investigate whether or not the accumulation of technical debt interest can be avoided by focusing on the quality of the new code. Specifically, we consider the generation of interest as a risk and present a metric (namely Interest Generation Risk Importance–IGRI) to quantify this risk. Furthermore, we validate this metric in two ways. Firstly, we confirmed with industry experts that the metric can be used to effectively prioritize technical debt items (artifacts that pose a more important risk to generate technical debt interest, would be more urgent for refactoring). Secondly we corroborated that adding new code reduces the risk of interest generation: if new code is less risky to generate interest, the sustainability of the system can be improved by the addition of clean new code.

Samenvatting

Technical Debt (Technische schuld - TD) is een krachtige metafoor die de snelkoppelingen vertegenwoordigt die in een software ontwikkelingsproject worden genomen, meestal om zakelijke doelen te bereiken, zoals een beperkte tijd of een beperkt budget. Technical Debt betreft veelal onzichtbare delen van het systeem (dus geen zichtbare kenmerken of defecten) die een nadelig effect hebben op de onderhoudbaarheid en evolueerbaarheid van software. Technische schuld manifesteert zich in zowel ontwerp- als implementatieconstructies, zoals suboptimale ontwerpbeslissingen, ontwerp- en code smells, gebruik van brutale technologieën, gebrek aan documentatie enz.

Technical Debt kan realistisch gezien niet worden geëlimineerd, maar het moet wel proactief worden beheerd om op een duurzaam niveau te worden gehouden. Het beheren van Technical Debt omvat een aantal activiteiten. In deze scriptie richten we ons op twee van deze activiteiten, die in de literatuur als de belangrijkste worden vermeld: terugbetaling en voorkomen. Zoals de naam al doet vermoeden, verwijst terugbetaling naar de sanering van de Technical Debt door het herwerken van de bovengenoemde ontwerp- en implementatieconstructies; denk bijvoorbeeld aan het refactoren van de broncode, het optimaliseren van een ontwerpbeslissing of het documenteren van stilzwijgende architectuurn kennis. Evenzo verwijst het voorkomen van Technical Debt naar het ervoor zorgen dat de onderliggende problemen zich in de eerste plaats niet voordoen; denk bijvoorbeeld aan het controleren op schendingen van de broncode, het ondersteunen van ontwerpbeslissingen om toekomstige onderhoudsoverhead te voorkomen of het eenvoudigweg toewijzen van voldoende middelen om het juiste te doen in plaats van de 'quick and dirty'-aanpak. In deze scriptie proberen we te begrijpen in hoeverre terugbetaling en preventie effectief zijn bij het beheersen van Technical Debt.

Om het genoemde probleem aan te pakken, moeten we eerst de evolutie van Technical Debt in de praktijk beter begrijpen. Specifiek moeten we onderzoeken of de technical debt in de loop van de tijd groeit of stabiel blijft, zowel op macroscopisch niveau als op het

niveau van individuele problemen en hun typen. Door een grootschalige casestudy hebben we geleerd dat in de meeste systemen die we hebben bestudeerd, er een aanzienlijke stijgende trend is wat betreft de omvang, het aantal problemen en de complexiteitsstatistieken van het project. Aan de andere kant neemt het bedrag aan technical debt per regel code (TD-dichtheid) af naarmate het project evolueert. De verbetering van de TD-dichtheid is een belangrijke bevinding, in die zin dat het in tegenspraak is met de 7e van Lehman's evolutiewetten, die stelt dat de kwaliteit van software in de loop van de tijd achteruitgaat. Dit roept de vraag op waarom: is het vooral te wijten aan aflossing of preventie? In de volgende delen van deze scriptie onderzoeken we specifiek of de TD-dichtheid kan worden verlaagd door stukjes code die lijden aan ontwerp- of code-inefficiënties (terugbetaling) te refactoren of door te streven naar de introductie van nieuwe code die geen of een zeer lage technical debt heeft (preventie).

We beginnen met het onderzoeken van het geval van refactoring. Concreet willen we nagaan of refactoring in de praktijk systematisch wordt toegepast, of dat het meer wishful thinking is. Recente literatuur heeft aangegeven dat refactoring in de praktijk nogal beperkt is, omdat ontwikkelaars vooral bezig zijn met het introduceren van nieuwe functies of het oplossen van bugs. Via een tweede grote case study over Open Source-software ontdekten we dat een groot percentage van de technical debt tijdens de evolutie wordt terugbetaald; voor de meeste projecten ligt het bevestigingspercentage echter onder de 30%. Dit houdt in dat refactoring niet zo effectief is bij het aanpakken van technical debt, aangezien meer dan 70% van de technical debt niet vaststaat.

Nadat we het geval van refactoring hebben bestudeerd, gaan we verder met het tweede geval: is het introduceren van nieuwe, schone code een betere manier om de genormaliseerde technical debt te verminderen? Hiertoe hebben we een derde case study uitgevoerd, waaruit bleek dat de nieuw ingevoegde code een hogere kwaliteit vertoont (in termen van TD-dichtheid), vergeleken met de kwaliteit van de systemen waarin de nieuwe code is geïntroduceerd en dit was consistent in alle onderzochte projecten. Het is dus mogelijk dat de algehele kwaliteit van het systeem kan worden verbeterd als de TD-dichtheid van nieuw ingevoegde code systematisch onder het systeemgemiddelde wordt gehouden, bijvoorbeeld door het gebruik van Quality Gates.

Nadat we hebben vastgesteld dat de dichtheid van technical debt kan worden verlaagd met behulp van nieuwe, schone code, onderzoeken we de impact van schone code op technical debt. In het bijzonder bestuderen we de impact van nieuwe code op de twee kernconcepten van technical debt: hoofdsom en rente. De hoofdsom verwijst naar tekortkomingen zoals code smells, overtredingen van codeerregels, design smells enz. die moeten worden verholpen. Daarentegen verwijst technical debt rente naar de extra onderhoudsinspanning en bijbehorende kosten die voortvloeien uit het bestaan van technical debt posten in een systeem op zich.

Uitgaande van de hoofdsom van de technical debt, bestuderen we hoe het fluctueert met

de evolutie, aangezien een softwaresysteem meestal groeit door de toevoeging van nieuwe code. Via onze vierde case study ontdekten we dat de introductie van problemen met technical debt als gevolg van het invoegen van nieuwe code tijdsafhankelijk is, voor meer dan de helft van de projecten. Hoewel dit kan worden geïnterpreteerd als een indicatie van de volwassenheid van het project, betekent zelfs voor deze stabiele projecten dat de afwezigheid van fluctuaties niet noodzakelijkerwijs de afwezigheid van enige trend betekent. Ten slotte zijn het aantal commits gemaakt in een bepaalde periode en de ervaring van ontwikkelaars niet gecorreleerd met het aantal geïntroduceerde technical debt problemen in het systeem (voor de meeste gevallen). Dit is contra-intuïtief, maar we vermoeden dat andere versturende factoren relevanter kunnen zijn, zoals: (a) de looptijd van het project; (b) de gewoonten van de ontwikkelaars; of (c) het specifieke type taken dat in elke periode wordt uitgevoerd.

Als we verder gaan met het tweede bestanddeel van technical debt, namelijk de rente, in het laatste deel van deze scriptie, onderzoeken we of de accumulatie van technical debt rente kan worden voorkomen door ons te concentreren op de kwaliteit van de nieuwe code. In het bijzonder beschouwen we het genereren van rente als een risico en presenteren we een maatstaf (namelijk Interest Generation Risk Importance – IGRI) om dit risico te kwantificeren. Bovendien valideren we deze statistiek op twee manieren. Ten eerste hebben we met branche-experts bevestigd dat de statistiek kan worden gebruikt om technical debt posten effectief te prioriteren (artefacten die een groter risico vormen om technical debt rente te genereren, zouden urgenter zijn voor refactoring). Ten tweede hebben we bevestigd dat het toevoegen van nieuwe code het risico op het genereren van interesse vermindert (als nieuwe code minder risicovol is om interesse te genereren, kan de duurzaamheid van het systeem worden verbeterd door het toevoegen van schone nieuwe code).

Contents

Samenvatting	ix
Acknowledgements	xvii
List of figures	xix
1 Introduction	1
1.1 Technical Debt and its Management	2
1.2 Clean New Code as a Form of TD Prevention	3
1.3 Research Design	3
1.3.1 Problem Statement	4
1.3.2 Design Science Framework	4
1.3.3 Problem Decomposition	5
1.3.4 Empirical Research Methodology	9
1.4 Overview of the Dissertation	9
2 The Evolution of Technical Debt in the Apache Ecosystem	11
2.1 Introduction	11
2.2 A Motivating Example	12
2.3 Study Design	16
2.3.1 Project Selection	17
2.4 Results and Discussion	17
2.5 Threats To Validity	23
2.6 Related Work	24
2.7 Conclusions and Future Work	26

3	How do developers fix issues and pay back technical debt in the Apache ecosystem?	27
3.1	Introduction	27
3.2	Related Work	28
3.3	Studying Technical Debt Evolution	31
3.3.1	Using SonarQube for Debt Estimation	31
3.3.2	Studying Multiple Revision of a System	31
3.3.3	Detecting Fixed Issues	32
3.3.4	Classifying Issues	33
3.4	Study Design	33
3.4.1	Project Selection	34
3.4.2	Issue Selection	34
3.5	Results and Discussion	37
3.5.1	How does the code TD issue fixing rate vary for different projects? (RQ ₁)	37
3.5.2	What is the fixing prevalence of the various code TD issue types? (RQ ₂)	38
3.5.3	How does the fixing rate vary for different code TD issue types? (RQ ₃)	40
3.5.4	How is the effort of paying back code TD distributed across the various types? (RQ ₄)	42
3.5.5	After how much time is code TD paid back? (RQ ₅)	46
3.6	Observations and Open Questions	47
3.7	Threats To Validity	48
3.8	Data Set and Replication	48
3.9	Conclusion and Future Work	49
4	Can Clean New Code reduce Technical Debt Density	51
4.1	Introduction	51
4.2	Related Work	53
4.2.1	Evolution of Code Smells	53
4.2.2	Refactoring Frequency and Impact	55
4.2.3	Quality Gates	57
4.3	Case Study Design	58
4.3.1	Goal and Research Question	58
4.3.2	Cases and Units of Analysis	59
4.3.3	Tracking the Types of Changes	60
4.3.4	Data Collection	62
4.3.5	Data Analysis	66
4.4	Results	68

Contents

4.4.1	Descriptive Statistics	68
4.4.2	Relation among the contribution of new / deleted / modified methods and change in system's $TD_{density}$ (RQ ₁)	71
4.4.3	Relation between Code Quality Practices and New Code Cleanness (RQ ₂)	74
4.5	Discussion	75
4.5.1	Interpretations of the Results	76
4.5.2	Implications for Practitioners	79
4.5.3	Implications for Researchers	79
4.6	Threats To Validity	80
4.6.1	Construct Validity	80
4.6.2	Reliability	81
4.6.3	External Validity	81
4.7	Conclusions	82
5	The Temporality of Technical Debt Introduction on New Code and Confounding Factors	87
5.1	Introduction	87
5.2	Related Work	90
5.2.1	Causes of Technical Debt Introduction	90
5.2.2	Evolution of Technical Debt	91
5.3	Background Information	92
5.4	Case Study Design	94
5.4.1	Research Questions	94
5.4.2	Cases and Units of Analysis	95
5.4.3	Data Collection	97
5.4.4	Data Analysis	98
5.5	Results	99
5.5.1	Fluctuation of TD introduction along evolution (RQ1)	99
5.5.2	Correlation of Fluctuation vs. Activity (RQ2.1) and Experience (RQ2.2)	101
5.6	Discussion	102
5.6.1	Interpretation of Results	102
5.6.2	Implications to Researchers and Practitioners	103
5.7	Threats to Validity	104
5.8	Conclusions and Future Work	105
6	The Risk of Generating Technical Debt Interest: A Case Study	111
6.1	Introduction	111
6.2	Related Work and Background Information	113

6.3	Assessing the Risk of Generating Interest	114
6.3.1	Technical Debt Interest	115
6.3.2	Technical Debt Interest Probability	116
6.4	Case Study Design	117
6.4.1	Research Goals and Questions	117
6.4.2	Cases and Units of Analysis	118
6.4.3	Data Collection	118
6.4.4	Analysis	120
6.5	Results	120
6.5.1	Ability of IGRI to Prioritize TD Artifacts	121
6.5.2	Relation of IGRI and new code	122
6.6	Discussion	125
6.7	Threats to Validity	125
6.8	Conclusions	126
7	Conclusions and Future Work	127
7.1	Answers to Research Questions	127
7.2	Ongoing and Future Work	129
7.2.1	Clean New Code vs Refactorings	129
7.2.2	Personalized assessment of code TD introduction	130
7.2.3	Investigating the industrial applicability of the 'clean new code' paradigm	130
	Bibliography	131

Acknowledgments

The current dissertation is the result of a challenging procedure and could not have been accomplished if I were not surrounded by many special people who supported me during this journey. Now that I am completing this project, I feel the need to thank all people that helped me to complete this PhD project.

First of all, I want to express my special thanks and sincere gratitude to my supervisors from the University of Groningen (Professor Paris Avgeriou and Professor Mircea Lungu) and the University of Macedonia (Professor Alexander Chatzigeorgiou and Professor Apostolos Ampatzoglou) for believing in me and giving me the chance to start my PhD. They were always inspiring, supporting me and their guidance throughout my research was priceless and necessary in order to complete this dissertation. I learned many things from them and they helped me to improve various aspects of my work routine in so many ways.

I would also like to express my gratitude to the members of the assessment committee: Prof. Clemente Izurieta, Prof. Antonio Martini, and Prof. Tijs van der Storm for their precious time, the valuable feedback that I received, and the attendance of my defense. I would also like to thank all the reviewers of my papers for their time and their valuable feedback on how to improve them.

Many thanks to all the members of the Software Engineering and Architecture group from the University of Groningen: Andrea Capiluppi, Anja Reuter, Ayushi Rastogi, Cezar Sas, Chen Yang, Christian Manteuffel, Daniel Feitosa, Darius Daniel Sas, Héctor Fabio Cadavid Rengifo, Jie Tan, Mohamed A.M. Soliman, Peng Liang, Ruiyin Li, Sara Mahdavi-Hezavehi, Sofia Charalampidou, Theo Theunissen, Tijs van der Storm, Vasilios Andrikopoulos, Xiao He, Yikun Li, Zaki Pauzi, and Zengyang Li, as well as the members of the Software Engineering group from University of Macedonia: Areti Ampatzoglou, Christina Volioti, Dimitrios Tsoukalas, Dimitrios Zisis, Elvira-Maria Arvanitou, Georgios Melas, Nikolaos Nikolaidis, Stratos Kourtzanidis, Theodoros Chaikalis, and Theodoros Maikantis, for their support, feedback, and all the activities that we enjoyed together.

Special thanks go to Chen Yang. He is by far the best office-mate. During my PhD we made a great team, we did many activities together and he was always providing me with valuable

feedback on my research when I was needing it. Also, I will never forget how great a host he was during my trip to China where I learned so many new things and I visited some really nice places.

During the period that I stayed in Groningen, I made a lot of friends and I met some really nice people. I would like to thank Prof. Kyriaki Fousiani and Pieter Minnigh. I am really glad that I met you and I always enjoy your company. I am also really happy that I met my two Cypriot friends Prof. Maria Ioannou and Prof. Lucy Avraamidou, I had a really good time hanging out with you. I would also like to thank all the employees of the University of Groningen for their support and for offering me a nice environment to pursue my PhD. Especially, many thanks to Franke Steffens for helping me with all the things that I had to do when I arrived at the University of Groningen.

I am also grateful to Prof. Eleni Koustriava, for all the things that she has done for me. She is always available for help, providing me with advice and strength to keep going. I am really glad I have met her.

Many thanks to my Uncle Antonios Digkas for his support, his suggestions and for always pushing me to study more and to strive for perfection in what I do.

I would also like to leave a testament of my love and gratitude to my Aunt Paschalina Digka, who passed away, suddenly, due to the COVID-19. Thank you for everything you have done for me, from the beginning of my life.

Last but not least, my family deserves the greatest thanks. I want to thank my Mother Evgenia Siakavara, my Father Ioannis Digkas, and my siblings: Dimitrios Digkas, Aikaterini Digka, and Philippos Digkas, as well as my brother's wife and child Konstantina Diakoglou and Ioannis Digkas (respectively), for their unconditional love and support.

Georgios Digkas
Thessaloniki
September 20, 2021

List of Figures

1.1	Problem Decomposition	6
2.1	The evolution of lines of code (blue, top), non commented lines of code (green, middle), and technical debt (red, bottom) in Apache Sling over the last 5 years. Technical debt represents effort to fix problems and is estimated in minutes.	13
2.2	The evolution of Normalized Technical Debt in Apache Sling over the last 5 years shows a clear decreasing trend	14
2.3	Trend results for Technical Debt	19
2.4	Trend results for normalized Technical Debt	20
3.1	Correlation between the size of a project (in number of files) and commit density	35
3.2	Distribution of resolved issues by priority type (the light gray background presents the total number of issues of that kind)	36
3.3	Number of open (light yellow) and closed (dark green) issues per project. The top part of the figure presents the sizes of the corresponding projects; the middle absolute numbers of open/closed issues; the bottom part presents percentages of open/closed issues	36
3.4	Distribution of fixing rates in examined projects for bins of size 0.10	37
3.5	Distribution of resolved issues by issue type for the 160 types of issues in this study	38
3.6	Fixing rate (horizontal axis) vs. Number of Issues for a given Issue Type . .	41
3.7	Histogram of the survival time for the 155K issues in this study with bins of 30 days	45
3.8	Distribution of survival time for top the ten issues (according to RQ ₂) . . .	46

4.1	Seeking the longest path between commit nodes	60
4.2	Contribution of new code on system's $TD_{density}$ (Motivating Example) . .	68
4.3	Distribution of the difference between the $TD_{density}$ of new methods (introduced in new classes) and the $TD_{density}$ of existing system, for all projects	70
4.4	Distribution of the difference between the $TD_{density}$ of new methods (introduced in existing classes) and the $TD_{density}$ of existing system, for all projects	71
4.5	Percentage of revisions in which a decrease in the system $TD_{density}$ co-occurred with a positive contribution in quality by new and modified methods	72
4.6	Distribution of the percentage of cleaner code commits for the two groups of projects, based on the existence of commit guidelines	74
4.7	Distribution of the percentage of cleaner code commits for the two groups of projects, based on the frequency of references to code quality in board meetings	75
5.1	TD Introduction Temporality	88
5.2	Demarcating Units of Analysis (sliding temporal windows)	98
5.3	Indicative project evolution	100
5.4	Project Fluctuations	100
5.5	Activity vs. TD Introduction – Dubbo project	101
5.6	Activity vs. TD Introduction – PDFBox project	102
5.7	Experience vs. TD Introduction – Left: Dubbo, Right: TinkerPop	102
6.1	Risk Assessment Matrix	114
6.2	Notion of hypothetical optimal among similar artifacts	116
6.3	Risk Ranking Consistency of IGRI and Perception of Stakeholders	122
6.4	Percentage of New Code and Risk of Producing Interest	123
6.5	Quality of New Code and Risk of Producing Interest	124

Chapter 1

Introduction

The Technical Debt (TD) metaphor was coined by Ward Cunningham (Cunningham 1992) in 1992 as follows: *“Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. Objects make the cost of this transaction tolerable. The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise”*. The metaphor was used to emphasize the trade-off between delivering immature source code and shortening the release date or time to market, by drawing a parallel between “shipping first time code” and “going into debt” (Cunningham 1992). In the recent years, technical debt has attracted the attention of many researchers as well as software practitioners. It is being used in practice especially as a means of communication between technical and management stakeholders. In the literature, there are many types of technical debt. Nicolli Alves et al. (Alves et al. 2014) proposed an ontology of the following 13 technical debt types: Architecture Debt, Build Debt, Code Debt, Defect Debt, Design Debt, Documentation Debt, Infrastructure Debt, People Debt, Process Debt, Requirement Debt, Service Debt, Test Automation Debt, Test Debt and a set of key indicators for each one of them. This study focuses exclusively on Code Technical Debt and therefore the results cannot be generalized to other types of technical debt.

Currently, there are different definitions of technical debt from different researchers as well as from prominent practitioners. This thesis relies on and uses the most well-accepted definition by researchers, which is the one from the Schloss Dagstuhl seminar of 2016 (Avgeriou et al. 2016). Namely: *“In software-intensive systems, technical debt is a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible. Technical debt presents an actual or contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability.”*

The rest of this chapter is organized as follows: In Section 1.1 we present the background information on the main technical debt concepts; in Section 1.2 we discuss the notion of writing clean new code; in Section 1.3, we elaborate on the research design of the thesis; finally, in Section 1.4 we give an overview of the rest of this dissertation.

1.1 Technical Debt and its Management

Software engineers often take shortcuts because of stringent delivery deadlines that have to be met or other forms of time pressure. Technical Debt reflects the additional maintenance effort that a developer or development team has to spend when a such shortcuts are taken. Two key terms of the technical debt metaphor which are being widely studied in this thesis are Principal and Interest. *Principal* corresponds to a monetary estimate of the effort that is required to address identified inefficiencies at the code, design or architecture level. *Principal* reflects the distance between the actual and the optimal level of software quality in an immature software artefact or the complete software system. *Interest* is a monetary estimate of the additional effort needed to maintain the software (implement new features and/or extend/maintain existing ones), because of its sub-optimal quality. If technical debt remains untreated, the software project accumulates interest and may reach a point where changes are too costly to perform or even impossible.

Technical Debt Management (TDM) is the engineering process that aims at retaining the amount of TD at acceptable levels, and according to Li et al. (Li et al. 2015) it consists of nine main activities. Among them, the two that are related to reducing the amount of TD are repayment and prevention. The secondary study by Li et al. (Li et al. 2015) defines **TD repayment** as the resolution or mitigation of TD in a software system and suggests a plethora of research approaches on TD repayment, compared to other TD management activities. These approaches are classified into seven categories, among which we present the two most prominent ones:

1. Refactoring: The application of changes to the code, design, or architecture of a software system in order to improve its internal quality, without however altering its external behaviour.
2. Rewriting: Writing from scratch the code that contains TD

Among these categories of technical debt repayment approaches, refactoring is the most used (Li et al. 2015) and widely studied in the literature. The majority of studies on refactoring concern code technical debt, mainly due to: (a) the availability of tools supporting the identification, measurement, and repayment of this type of TD; (b) the fact that code TD is concrete and easy to understand; and (c) the experience that practitioners have on using code analysis and code refactoring tools in their every-day work (Li et al. 2015). However, other types of software artifacts can also be refactored (i.e. design artifacts like design patterns, object-oriented database schemas, and software architectures; and software requirements) (Mens and Tourwé 2004).

On the other hand, **TD prevention** refers to averting potential TD from being incurred (Li et al. 2015). According to Li et al., such activities are classified in the following categories:

1. Development process improvement: The improvement of current development processes in order to prevent the occurrence of certain TD types.
2. Architecture decision making support: The identification of where TD is located and the evaluation of different architecture design alternatives in order to choose the option with less potential TD.
3. Lifecycle cost planning: The development of cost-effective plans that consider the overall minimization of the system TD, i.e. throughout its lifecycle.
4. Human factors analysis: The creation of a culture that minimizes the unintentional TD caused by human factors, e.g., indifference or ignorance.

In the next section, we present a specific approach for TD prevention, namely writing “Clean New Code”, which touches upon all (but the 2nd) categories.

1.2 Clean New Code as a Form of TD Prevention

Ward Cunningham (Cunningham 1992) vividly pointed out that the root cause of technical debt is ‘*not quite right code*’. The most obvious choice for preventing the introduction of technical debt is to write **Clean New Code** in the first place. This thesis posits that writing clean new code is an approach that is more appealing to software developers compared to refactoring which is an after-the-fact process.

Numerous empirical studies on the application of refactoring revealed that refactorings are scarcely applied. However, there is evidence that several software systems retain or even improve their level of quality along their evolution, even if systematic refactoring does not take place. One plausible explanation for this reversal of the software-ageing phenomenon is that the development of new software artifacts is at a level of quality that is above the average. This clean new code paradigm can be encouraged or even imposed by the use of Quality Gates which essentially do not allow code that does not satisfy certain quality criteria to be committed to a software repository.

While refactoring has been the focus of various studies, technical debt prevention through the writing of clean code has not. While ensuring that a single commit has low or even zero technical debt will contribute to a very small extent to the overall system’s quality, the aggregate effect of cleaner code can be efficient in managing technical debt in the long term.

1.3 Research Design

In this section the research design of the thesis is presented. The problem statement, that is addressed in this thesis, is introduced in section 1.3.1. In section 1.3.2 the employed research

methodology is presented, whereas in section 1.3.3 the research questions that this thesis deals with are listed. In section 1.3.4 the used empirical research methods are presented.

1.3.1 Problem Statement

Software change is inevitable so as for the product to remain up-to-date and solve relevant problems for customers. The development activity that deals with accommodating software changes is termed software maintenance and evolution, and it can be one of the costlier activities in the software life-cycle (Van Vliet et al. 2008). Software maintenance and evolution activities are initiated by either the customer (add new features, bug fixing, environmental changes, etc.) or the quality assurance team (e.g., preventive maintenance), and is passed on to the development team in the form of maintenance tickets. These tickets are assigned to developers to resolve, which subsequently attempt to spot the pieces of code that need to be maintained, and then apply the change (Kosti et al. 2018).

The effort that is spent in the maintenance and evolution activity can be substantially increased, if the software code-base is not well implemented / designed, i.e. it suffers from Technical Debt (see Section 1.1). Nevertheless, maintenance and evolution activities must be fulfilled as soon as possible, since otherwise, the users and/or customers will be dissatisfied, potentially leading to the software being abandoned and incurring financial loss. Therefore, it becomes crucial that the system quality is maintained and can facilitate changes without excessive effort. To this end, the development team should monitor the evolution of technical debt, and retain it at acceptable levels. As mentioned in the previous sections, there are two Technical Debt Management activities that can help in reducing TD: repayment and prevention. The goal of this project is to explore the aforementioned two activities, namely repayment through the activity of software refactoring and prevention through the activity of writing clean new code. Summarizing the above, this thesis addresses the following problem:

The code Technical Debt (TD) that exists in a software system is detrimental to the maintenance and evolution of that system, to the extent that the system may become unsustainable. Software engineers can either repay the TD through refactoring the code, or they can prevent the TD from occurring in the first place, through writing clean new code. Which of the two ways is followed in practice and how?

1.3.2 Design Science Framework

The research project, documented in this dissertation, adopts the design science framework as described by (Wieringa 2010). Design Science was initially introduced by (March and Smith 1995) aiming to assist stakeholders create or improve “things” in their technological solutions, that serve human purposes.

In Design Science there are two main components aiming to define and explore the problem of interest: the practical problems and the knowledge questions. A practical problem is

defined as “a difference between the way the world is experienced by stakeholders and the way they would like it (the world) to be” (Wieringa 2010). The analysis of a practical problem and the evaluation of a proposed solution are both knowledge questions, which usually emerge through repetitive rounds of activities, known as design cycles. A knowledge question is defined as “a difference between current knowledge of stakeholders about the world and what they would like to know” (Wieringa 2010). Knowledge questions are answered by using analytical or empirical research methods. The two are seamlessly nested to develop the artefact that was the initial target of the Design Science project. The workflow addressed in the design science framework is well-suited for describing long-term research such as PhD projects, since it facilitates the evolution of research questions and solutions at the same time. In particular, it suggests the decomposition of an initial problem statement into practical problems and knowledge questions which become more concrete as the researcher gains more domain knowledge, acquires a better understanding of the problem and progresses with the research solutions.

1.3.3 Problem Decomposition

In this section, we present the practical problems and knowledge questions addressed in this thesis, and how each one is followed up by another. Figure 1.1 depicts the knowledge questions (KQ) and practical problems (PP). We refer to both practical problems and knowledge questions as research questions. The main research questions are labeled with numbers from one to five. The research sub-questions are numbered with lowercase letters.

In order to address the stated problem of preventing or repaying technical debt, we first need to understand better the evolution of technical debt in practice. Specifically we need to investigate whether technical debt grows or remains stable over time, which types of technical debt appear the most frequently and how expensive it is to fix them (KQ1). To this end, we have conducted a case study on sixty-six Java open-source software projects by the Apache Software Foundation, attempting to answer the following 4 RQs.

RQ_{1.a}: How does the code technical debt of the open-source systems in the Apache ecosystem evolve over time?

RQ_{1.b}: How does the normalized code technical debt of the open-source systems in the Apache ecosystem evolve over time?

RQ_{1.c}: What are the most frequent types of code technical debt in the studied ecosystem?

RQ_{1.d}: What are the most costly to fix types of code technical debt in the studied ecosystem?

The results of this study helped us to understand in detail how technical debt evolves. They have confirmed the intuition of developers that technical debt increases over time for the majority of the systems, similarly to source code metrics. However, the density of technical debt, i.e. technical debt normalized to the size of the system actually decreases over time

Figure 1.1: Problem Decomposition

in the majority of the systems under investigation. This is a rather optimistic finding, as it implies that technical debt per line of code is reduced along system evolution. This begs the question why this happens. A fraction such as the normalized TD, can be decreased either by increasing the denominator (namely LoC) or by decreasing the numerator (namely TD). Technical Debt can be decreased either by refactoring pieces of code suffering from design or code inefficiencies or by striving to introduce new code that has zero or very low technical debt. We investigate both cases in the following questions.

We begin by looking into the case of refactoring. Specifically, we want to find out whether refactoring is systematically applied in practice (KQ2), or whether it is more wishful thinking. Recent literature has indicated that refactoring is rather limited in practice (Chatzigeorgiou and Manakos 2014, Peters and Zaidman 2012, Arcoverde et al. 2011, Tufano et al. 2017, Tufano et al. 2016), as developers are mostly busy with introducing new features or fixing bugs. To answer KQ2, we focused on the amount and types of issues that are fixed through refactoring, as well as the effort and time this takes. Thus, we defined the following five RQs.

RQ_{2.a}: How does the code TD issue fixing rate vary for different projects?

RQ_{2.b}: What is the fixing prevalence of the various code TD issue types?

RQ_{2.c}: How does the fixing rate vary for different code TD issue types?

RQ_{2.d}: How is the effort of paying back code TD distributed across the various types?

RQ_{2.e}: After how much time is code TD paid back?

In order to answer these RQs, we conducted a case study, in which we analyzed the evolution of fifty-seven Java open-source software projects by the Apache Software Foundation at the temporal granularity level of weekly snapshots. In particular, we focused on the amount of technical debt that is paid back through refactoring and the types of issues that are fixed. The findings reveal, among others, that a large percentage of technical debt is paid back during evolution; however, for the majority of the projects, the fixing rate is below 30%. This entails that refactoring is not as effective in addressing technical debt as more than 70% of the technical debt items are not fixed.

After studying the case of refactoring, we move on to the second case: does introducing new, clean code help with reducing normalized technical debt (KQ3)? In other words, we want to study the benefits of writing clean new code as a means of sustaining a project's long-term quality. To this end, we shift our focus on the quality of newly inserted Code, in terms of the technical debt of new code introduced in each commit in the software history. Specifically, we compare the $TD_{density}$ of new code with that of existing code, and we investigate which of the three major types of code changes (additions, deletions and modifications) is primarily responsible for changes in the evolution of $TD_{density}$. Furthermore, we study whether there is a relation between code quality practices and the 'cleanness' of new code. Thus, we formed the following two RQs.

RQ_{3.a}: Among the three major types of code changes (insertion, deletion and modification) which is primarily responsible for changes in code $TD_{density}$?

RQ_{3.b}: Is the frequency of commits in which new code is cleaner compared to exiting code, associated with the existence of practices targeting code quality?

This study aims to investigate if the newly inserted code exhibits higher quality (in terms of $TD_{density}$), compared to the existing quality of the system. In other words, we examine, if it is possible to improve system's overall quality, if the $TD_{density}$ of newly inserted code is systematically kept below the system's average.

Supposing that the density of technical debt (which again is the amount of technical debt per line of code) can be decreased using new, clean code, we investigate further the impact of clean code on technical debt. Specifically, as mentioned in Section 1.1, the concept of technical debt is established in the literature as a composition of two parts: the *principal* and the *interest* of technical debt. The principal refers to shortcomings like code smells, violations of coding rules, design smells etc. that need fixing. In contrast, technical debt interest refers to the additional maintenance effort and associated costs that stem from the very existence of technical debt items in a system. The following two questions study the impact of new code on technical debt principal and interest respectively.

Starting from technical debt principal, we aim at studying how it fluctuates along evolution (KQ4), since a software system grows mostly through the addition of new code. This can shed light on whether technical debt is introduced in an almost flat rate, whether large volumes of technical debt are introduced in infrequent timestamps, or whether large volumes of TD are frequently introduced along evolution. Consequently, in the second and third case, we would need to revise our technical debt management techniques to focus on those times where large TD is introduced to prevent or limit the phenomenon. Furthermore, it is interesting to study whether the temporality of technical debt introduction relates to the development team’s workload in a given period, as well as to the experience of the development team. Thus, we ask the following two RQs:

RQ_{4.a}: Does the number of code Technical Debt Issues introduced by new code fluctuate along evolution?

RQ_{4.b}: Does the amount of code Technical Debt introduced by new code correlate to the activity or the experience of the developers?

To answer these questions, we performed a case study on thirty-five open-source software projects from two well-known ecosystems (Apache and Eclipse), and inspected the number of Technical Debt issues introduced in 6-month sliding temporal windows.

Moving on to the second constituent of technical debt, i.e. the *interest*, we investigate whether or not the accumulation of technical debt interest can be avoided by focusing on the quality of the new code (KQ5). This is of paramount importance, as the generation of technical debt interest may make the system very hard to change, or sometimes even impossible. Thus, in the final study, we consider the generation of interest as a risk and come up with a metric (namely Interest Generation Risk Importance–IGRI) to quantify this risk (RQ_{5.a}). For the purposes of this study, in order to measure the technical debt interest, we rely on the methodology that was proposed and validated by Ampatzoglou et al (Ampatzoglou et al. 2018). Furthermore, to validate IGRI we used a 2-step approach. First, we explore whether the metric can be used to effectively prioritize technical debt items (artifacts that pose a more important risk to generate technical debt interest, would be more urgent for refactoring). Second, we investigate if adding new code reduces the risk of interest generation (if new code is less risky to generate interest, the sustainability of the system

can be improved by the addition of clean new code). Formally, we ask the following RQs:

RQ_{5.b}: Is IGRI able to prioritize TD items similarly to an expert?

RQ_{5.c}: Does new code pose a lower risk (in terms of IGRI) for generating TD interest?

1.3.4 Empirical Research Methodology

To answer the knowledge questions of this project, the empirical software engineering paradigm has been used (Tichy and Padberg 2007). In particular, in the course of the project, the case study method has been applied several times. Case studies investigate “*a contemporary phenomenon within its real-life context, especially when the boundaries between phenomenon and context are not clearly evident*” (Yin 2017). The cases of the thesis were open source software projects, from well-known ecosystems, such as the Apache Software Foundation or the Eclipse Foundation. Regarding data collection we relied on the analysis of work artifacts (Lethbridge et al. 2005). By taking into consideration that for answering the majority of our knowledge questions a vast amount of data needed to be collected, a large portion of this project was devoted to developing tools. The most prominent outcome of this endeavor was the development of the “*New Code Analysis*” feature of the SDK4ED platform (see platform.sdk4ed.eu). An overview of the empirical research methods that were used for answering each knowledge question is provided in Table 1.1, which also maps the sections in this dissertation that elaborate on the study designs.

1.4 Overview of the Dissertation

The main body of this dissertation contains five chapters. Table 1.1 presents the knowledge questions, the practical problems, and the chapters, in which they are addressed.

Chapters 2 to 6 are based on scientific journal or conference articles that are either published or under revision. In all the publications, the PhD student was the first author and main contributor; other authors include the 2 supervisors as well as academic collaborators. In the following, each chapter is briefly outlined.

Chapter 2 is based on a paper published in the 2017 European Conference on Software Architecture (ECSA) (Digkas et al. 2018). That study was the first one that differentiated the evolution of the accumulated technical debt from $TD_{density}$ (normalized by LoC), leading to further exploring how $TD_{density}$ is being reduced along software evolution. ECSA is one of the top venues in the software architecture community.

Chapter 3 is based on a paper published in the 2018 International Conference on Software Analysis, Evolution and Reengineering (SANER) (Digkas et al. 2018). This study focused on TD reduction through refactoring, by exploring the fixing of Technical Debt Issues along evolution, instead of introducing new ones. SANER is considered as one of the top-2 venues in the software maintenance community.

Table 1.1: Details of research methods (case studies) used to answer the knowledge questions

	Knowledge Question	Empirical Method	Projects	Subjects	Chapter	Section
RQ ₁	How does code Technical Debt evolve over time?	Case Study	66	Apache	Ch. 2	2.3
RQ ₂	Is code TD repayment an established approach in OSS development?		57	Apache	Ch. 3	3.4
RQ ₃	Can practitioners control code TD along evolution by writing clean new code?		27	Apache Eclipse	Ch. 4	4.3
RQ ₄	Is the amount of code TD principal introduced by new code a temporal phenomenon?		47	Apache Eclipse OSS	Ch. 5	5.4
RQ ₅	Can practitioners control code TD Interest along evolution by writing clean new code?		1	Industrial	Ch. 6	6.4

Chapter 4 is based on a published article in the IEEE Transactions on Software Engineering (TSE) (Digkas, Chatzigeorgiou, Ampatzoglou and Avgeriou 2020). This paper introduces the concept of quality gates for TDM, it provides empirical evidence on the usefulness of writing clean new code, and delivers a tool for the above. It needs to be noted that TSE is the top journal in the software engineering community.

Chapter 5 is based on a paper under revision in Software Quality Journal (SQJ) (Digkas, Ampatzoglou, Chatzigeorgiou and Avgeriou 2020). This study explores the evolution of TD principal, and investigates if it is a temporal phenomenon. The temporality of TD introduction is contrasted to developers' experience and commit activity, as tentative factors of such temporality.

Chapter 6 is based on a paper published in the Springer Nature Computer Science (Digkas et al. 2021). This study builds upon the previous one and explores the relation of clean new code to TD interest. Incurring interest is considered as a risk, and a novel interest metric is introduced and validated.

Published as:

G. Digkas, M. Lungu, A. Chatzigeorgiou, and P. Avgeriou, “The evolution of technical debt in the apache ecosystem,” in *European Conference on Software Architecture (ECSA)*, Canterbury, United Kingdom, Springer, 2017, pp. 51-66, doi: 10.1007/978-3-319-65831-5_4.

Chapter 2

The Evolution of Technical Debt in the Apache Ecosystem

Abstract

Software systems must evolve over time or become increasingly irrelevant says one of Lehman’s laws of software evolution. Many studies have been presented in the literature that investigate the evolution of software systems but few have focused on the evolution of technical debt. In this paper we study sixty-six Java open-source software projects from the Apache ecosystem focusing on the evolution of technical debt. We analyze the evolution of these systems over the past five years at the temporal granularity level of weekly snapshots. We calculate the trends of the technical debt time series but we also investigate the lower-level constituent components of this technical debt. We aggregate some of the information to the ecosystem level.

Our findings show that the technical debt together with source code metrics increase for the majority of the systems. However, technical debt normalized to the size of the system actually decreases over time in the majority of the systems under investigation. Furthermore, we discover that some of the most frequent and time-consuming types of technical debt are related to improper exception handling and code duplication.

2.1 Introduction

The technical debt (TD) metaphor was coined by Ward Cunningham in 1992 as:

“Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. Objects make the cost of this transaction tolerable. The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise”.

technical debt is present in both industrial software as well as in open-source projects. In industrial settings the tight deadlines are pushing the software engineers and the developers

to compromise the quality of the system and take shortcuts in order to release a product as soon as possible. In open-source settings, the self-imposed deadlines of the developers working towards delivering their contributions to the community or the lack of processes regarding quality assurance might lead them to take similar shortcuts.

Taking all these shortcuts increases the change- and fault- proneness of the systems and aggravates long-term understandability, re-usability, reliability, efficiency, security, and maintainability. While it has been empirically proven that these shortcuts affect negatively the project's quality, completely eliminating technical debt from a system is undesirable as the investment to reduce technical debt would be extremely inefficient.

Although there has been extensive research with respect to technical debt (Li et al. 2015) and there exists even a dedicated international forum for research on the topic (the MTD workshop) there is a lack of empirical evidence regarding the occurrence and evolution of technical debt in the open-source systems.

Moreover, technical debt has also not been studied before in software ecosystems. Software ecosystems are groups of software projects that are developed and co-evolve in the same environment (Manikas 2016, Lungu 2009). Such projects can share code, they might depend on one another, and are often built on similar technologies and with similar processes.

In this paper we conduct an empirical study of sixty-six open-source software projects of the Apache ecosystem. We chose to analyze open-source software projects of the Apache Software Foundation because it is one of the biggest communities which provide software products for the public good and its projects are highly appreciated and used.

Structure of the paper. The rest of the paper is organized as follows: Section 2.2 motivates our study through an analysis of an Apache project. Section 2.3 presents the methodology and the design of the study. Section 2.4 presents the results of our empirical study and Section 2.5 discusses the threats to its validity. Section 2.6 presents the related work and Section 2.7 concludes the paper.

2.2 A Motivating Example

Apache Sling¹, one of the most popular projects in the Apache ecosystem, is an open-source Web framework for the Java platform designed to create content-centric applications on top of a JSR-170-compliant content repository such as Apache Jackrabbit. The project represents a significant community effort: at the moment of writing this article, the system has more than a dozen contributors who have commit rights to the main repository; these committers have contributed more than twenty thousand commits over the years.

Imagine we are the developers of the Apache Sling. We decide to analyze it to learn about the evolution of its technical debt. To measure the technical debt and extract other metrics

¹<https://sling.apache.org/>

we decide to use an industrial strength tool. SonarQube (Campbell and Papapetrou 2013) is an open-source tool for continuous inspection which features dashboards, rule-based defect analysis, and build integration. It supports various languages, including Java, C, C++, C#, PHP, and JavaScript.

SonarQube employs the Software Quality Assessment based on Lifecycle Expectations² method for estimating the time required to fix the technical debt (Letouzey and Ilkiewicz 2012). Technical debt evolution estimation using SonarQube is time consuming since when recomputing it for a new version of the system, no matter how small the difference between the two versions (even a single commit), the tool can not analyze only the differences, but instead, has to do the entire computation for the entire system again³. This means that the time necessary to analyze the history of a system is proportional to the number of versions of the system that are to be analyzed.

Figure 2.1: The evolution of lines of code (blue, top), non commented lines of code (green, middle), and technical debt (red, bottom) in Apache Sling over the last 5 years. Technical debt represents effort to fix problems and is estimated in minutes.

To drive this evolutionary analysis we do not use the graphical UI of the tool but rather we develop a program that interfaces with the API of the tool in order to compute the entire battery of analyses that SonarQube supports on that version, including technical debt related

²Software Quality Assessment based on Lifecycle Expectations

³This problem is not exclusive to SonarQube. We are not aware of any analysis tools that perform complex, system-level analysis without re-analyzing the entire system when presented with a new version.

analysis.

However, since analyzing all commits of the Sling project (which are more than twenty thousands) is not feasible as we explained earlier, we analyze snapshots of the system at one week intervals. More precisely we find the last commit in a given week, and we run the analysis on that version.

Figure 2.1 presents the evolution of the Apache Sling project over the last five years in terms of two metrics:

1. Lines of code (blue, topmost series) and non-comment lines of code (green, middle series)
2. Software Quality Assessment based on Lifecycle Expectations Index which is the tool estimated technical debt (red, bottom series) in minutes

The most salient observation in the Figure 2.1 is that the amount of measured technical debt as measured by the Software Quality Assessment based on Lifecycle Expectations index grows in parallel to the magnitude of the system as measured in lines of code. This is indeed not surprising as it is well known that as systems age their architecture erodes (Perry and Wolf 1992). Moreover, it is reasonable to assume that the absolute number of identified inefficiencies increases with the amount of functionality delivered by the system.

Figure 2.2: The evolution of Normalized Technical Debt in Apache Sling over the last 5 years shows a clear decreasing trend

Table 2.1: The most frequently occurring types of issues in Apache Sling. The number in parenthesis is the percentage of the total violations detected

	Issue	Count
1	The diamond operator (“<>”) should be used	2,888 (10.75%)
2	String literals should not be duplicated	2,875 (10.70%)
3	Generic exceptions should never be thrown	1,856 (6.90%)
4	Control flow statements should not be nested too deeply	1,345 (5.00%)
5	Exception handlers should preserve the original exceptions	1,215 (4.50%)

Table 2.2: The most costly to fix types of issues in Apache Sling. The starred issues did not appear in the previous table

	Issue	Time (minutes)
1	String literals should not be duplicated	39,234 (13.8%)
2	Generic exceptions should never be thrown	37,120 (13.0%)
3	*Source files should not have any duplicated blocks	30,440 (10.7%)
4	*Cognitive complexity of methods should not be too high	16,061 (5.6%)
5	Control flow statements should not be nested too deeply	13,450 (4.7%)

To exclude the possibility that the growth of technical debt is correlated with the growth of the system, we compute the evolution of the *size-normalized technical debt* – that is, the technical debt normalized to the number of lines of code in the system. Figure 2.2 shows that the normalized technical debt is actually decreasing.

Open Questions How this project compares with other similar ones? Is the growth of technical debt that our system exposes normal? Is the fact that the normalized technical debt decreases over time exceptional?

The Components of Technical Debt

The technical debt estimation is based on what SonarQube calls “rule violations” or *issues*. The tool detects the violations of a large variety of rules for software quality. These issues are classified by the tool into three categories: bugs, violations, and code smells. The version 6.2 of the tool that we used distinguishes among 397 Java rules. It divides them in the following 3 types namely: Bug (150 rules), Vulnerability (31 rules), and Code Smell (216). Table 2.1 shows the top 5 most frequent issues that are being violated in the case-study system.

Besides listing issues, SonarQube also estimates the time required to fix them. Table 2.2 shows the top 5 most time consuming issues as estimated by the tool.

The two tables show that the most frequent violations are not necessarily the most time consuming to fix: issue 1 in Table 2.2 was ranked lower in Table 2.1, and issues 3 and 4 did not even appear in that table. It is also interesting to see that some of the detected problems are quite low-level (e.g. string literal duplication) while others are relevant for the higher level architecture of the system (e.g. a seemingly absent exception policy, large scale code duplication).

Open Questions Are the relative frequency and effort required to fix these issues are characteristic to the Apache Sling project or they are more generally characteristic to Java systems? Is this uneven distribution of effort towards some issues specific or generic?

2.3 Study Design

Inspired by the open questions presented in the previous section, the *goal* of our study is then, to analyze the evolution of OSS projects in the Apache ecosystem for the *purpose* of understanding and investigating the accumulation of TD and the evolution of source code metrics. More specifically, our study aims at addressing the following four research questions (RQs):

RQ₁: *How does the code technical debt of the open-source systems in the Apache ecosystem evolve over time?*

The motivation for this question is to investigate the evolution of TD as it is generated by a widely acknowledged tool for a large set of OSS projects belonging to the same ecosystem.

RQ₂: *How does the normalized code technical debt of the open-source systems in the Apache ecosystem evolve over time?*

Because the amount of TD might be related to the size of the code base this research question aims at investigating the evolution of TD when normalized over the size of each system.

RQ₃: *What are the most frequent types of code technical debt in the studied ecosystem?*

The motivation for this question is to validate whether developers incur specific types of debt or not

RQ₄: *What are the most costly to fix types of code technical debt in the studied ecosystem?*

Since the effort required to repay TD varies among violations the goal of this question to obtain an insight into the actual effort to eliminate the most frequent sources of TD.

It is for brevity, that in the research questions and the rest of the paper we talk about *technical debt* but we clearly mean *technical debt as estimated by SQALE method implemented in the SonarQube tool*. The evolutionary study of technical debt as measured and estimated with other tools falls outside of our intended scope for this study.

2.3.1 Project Selection

The context of the study is the evolution of the Java open-source software projects developed by the Apache Software Foundation. Since the analysis we perform is computationally intensive we limit our study to a sample of sixty-six randomly selected Java projects from the ecosystem⁴. These represent more than a quarter of the Java projects in Apache. Table 2.3 presents the analyzed systems together with statistics about their magnitude.

We used the Apache Software Foundation Index⁵ in order to randomly select the projects that we analyzed. We used three inclusion criteria in order to decide whether we should analyze a project or not. We chose projects in which the main programming language is Java, have at least two years of evolution and are still active at the beginning of 2017. All the analyzed projects use git as version control system and they are hosted on GitHub, whence we cloned them.

The range in terms of weeks of evolution spans from 127 weeks to 767 weeks. We chose to analyze the last 5 years (260 weeks) of the evolution of the projects. The range of the number of classes for the first analyzed commit is from 0 to 7,040 and for the last analyzed commit from 35 - 8,295. At the same time the NCLOC for the first commit ranges from 0 to 450,186 and for the last commit from 5,197 to 635,020.

2.4 Results and Discussion

This section reports the analysis of the results achieved in our study and aims at answering the four research questions formulated in Section 2.3. A replication kit is available online at <https://github.com/digeo/evolution-of-td-in-apache>.

RQ₁: How does the code technical debt of the open-source systems in the Apache ecosystem evolve over time?

To answer RQ₁, for each project in the analyzed ones we created a weekly time series with the accumulation of technical debt. For each series we performed the Mann-Kendall test. The purpose of the Mann-Kendall (MK) test is to statistically assess if there is a monotonic

⁴The ecosystem contains projects written in more than 20 languages, but the majority of the projects is written in Java

⁵<https://projects.apache.org/projects.html?language#Java>

Table 2.3: The list of projects included in the study

Project	NCLOC	classes	Project	NCLOC	classes
sling	425,831	6,058	opennlp	62,141	998
zookeeper	74,898	948	chukwa	42,734	577
tomcat60	180,766	1,676	tapestry-5	157,911	3,266
jspwiki	57,967	555	manifoldcf	209,190	1,824
directory-shared	197,377	1,611	crunch	52,564	1,025
cayenne	232,876	3,818	jena	444,414	5,970
commons-collections	61,637	741	oodt	128,875	1,810
openjpa	431,915	5,358	sis	205,367	2,204
mina	23,633	442	commons-csv	5,197	35
poi	367,828	3,907	commons-vfs	33,315	427
nutch	51,738	639	falcon	122,277	1,015
commons-lang	74,849	569	aurora	68,894	1,156
commons-io	29,267	271	jclouds	340,647	6,950
httpclient	61,657	685	helix	81,729	1,060
wicket	211,627	4,175	struts	152,296	2,341
batik	191,790	2,590	cxf	635,020	8,295
roller	53,540	603	knox	72,188	1,177
maven	80,161	1,061	stratos	119,243	1,506
commons-cli	6,859	54	phoenix	273,435	2,134
wss4j	109,259	782	commons-math	186,584	1,685
pdfbox	136,997	1,337	tomcat80	317,555	3,425
aries	181,779	2,710	nifi	354,044	3,954
jmeter	124,358	1,408	vxquery	45,369	751
maven-surefire	58,107	1,248	zeppelin	81,218	982
commons-validator	15,930	159	polygene-java	159,748	4,500
stanbol	160,713	1,875	groovy	168,705	2,099
sqoop	76,273	837	apex-core	73,029	1,086
flume	84,882	1,000	apex-malhar	166,972	2,682
rampart	24,729	278	brooklyn-library	40,387	629
kafka	120,995	1,644	beam	199,476	3,631
giraph	97,952	1,870	tomcat85	306,473	3,397
oozie	159,043	1,325	incubator-hivemall	51,984	666
tomcat	303,901	3,428	qpuid-proton-j	38,055	613

upward or downward trend of the variable of interest over time. A monotonic upward (downward) trend means that the variable consistently increases (decreases) through time, but the trend may or may not be linear.

The null hypothesis (H_0) is that there is no monotonic trend and the alternative hypothesis (H_a) is that a monotonic trend is present. The value of the significance level (alpha error rate) is 0.01 ($\alpha = 0.01$).

Figure 2.3: Trend results for Technical Debt

We run MK test for each one of the analyzed systems. Figure 2.3 visually summarizes the results by presenting the Z values for the analyzed systems. If the Z value is above (below) the horizontal grey line, it indicates that an increasing (decreasing) trend is present.

The Figure 2.3 shows that in most of the projects, there is a monotonic upward trend of the technical debt over time.

RQ₂: How does the normalized code technical debt of the open-source systems in the Apache ecosystem evolve over time?

To address RQ₂ we extracted two series for each project. The first one contains data for the accumulation of technical debt and the second the number of the lines of code. Then, we divided the two series, namely: the technical debt series with the lines of code series to obtain what we call the *normalized technical debt* time series.

Figure 2.4: Trend results for normalized Technical Debt

Finally, for each normalized technical debt series we performed again the Mann-Kendall test. Figure 2.4 presents the Z values for the analyzed systems using the same conventions as before. It shows that:

1. For seven systems (approx. 10%), there is no clear trend (values between the two grey lines)
2. For eleven systems (approx. 20%), the normalized technical debt increases with time (values above the top grey line)
3. for the majority of the systems, the normalized technical debt decreases over time (values below the low grey line)

We find the third result from above encouraging. Indeed, one possible explanation is that the developers of these systems are concerned with paying back the technical debt. This is plausible considering that the systems under analysis are some of the most successful open-source systems and are regarded as high quality projects by the open-source community.

However, another possible explanation could be related to the different phases through which a system evolves; as the system moves towards the maintenance phase, the changes to the system will tend to be smaller such as patches and bug fixes, and thus, less likely to introduce technical debt.

Based on the answer to this research question, we realize that Apache Sling, the system we discussed earlier, was not special in the fact that its normalized technical debt was decreasing. But this is not surprising now, since we see that this is the case with the majority of the systems in the ecosystem, and we have picked Sling at random.

RQ₃: What are the most frequent types of code technical debt in the studied ecosystem?

To answer this question we summed up all open issues from all the analyzed systems. We only look at the issues that are still open in the last analyzed commit.

To gather insight into the distribution of the various types of issues across the different systems, we use the Gini coefficient. The Gini coefficient is a statistical measure of the degree of variation or inequality represented in a set of values, used especially in analyzing income inequality. Its value is between 0 and 1. A low coefficient is indicative for a uniform distribution in the analyzed values, while a high coefficient is indicative of a very skewed distribution.

Since there are more than a hundred types of issues we do not to present all of them but we limit the presentation to the top ten most frequent ones. The replication kit available online contains the full table of issues in the ecosystem.

Table 2.4 shows the ten most frequent types of issues encountered in the projects that we analyzed. By analyzing the information in the table, we observe that:

- The top 10 most frequent rule violations account for more than 40% of the issues in the systems. This hints at the fact that, if automated tool support would be developed for these issues, that would make a big difference.
- The most frequent issue is also the most easily remedied, since all the modern IDEs provide an Extract Constant/Variable refactoring. In fact, a recent study showed that Extract Constant/Variable is one of the most popular refactoring developers actually in practice (Murphy-Hill et al. 2011). This hints at the possibility that developers are not aware of the literal duplication and that future IDEs could auto-detect and suggest the removal of the problem.
- If we add up the two rules in the list that refer to exception handling, they are more frequent than the most frequent issue. This is a sign that exception handling in Java is still not being approached with sufficient discipline. Also this is a much higher-level abstraction than some of the other frequent issues.
- Code duplication, is another type of high-level, potentially architectural problem. It is not very frequent (2.4% of the issues pertain to it) but it has a very low Gini index, which means, it is very equally distributed among the analyzed systems.

Table 2.4: The ten most frequent types of technical debt in the Apache ecosystem

#	Issue	Count		Gini
1	String literals should not be duplicated	48,474	(7.0%)	.31
2	The members of an interface declaration or class should appear in a pre-defined order	38,756	(5.6%)	.43
3	Exception handlers should preserve the original exceptions	33,467	(4.8%)	.38
4	The diamond operator (“<>”) should be used	30,659	(4.4%)	.55
5	Generic exceptions should never be thrown	29,393	(4.2%)	.47
6	Statements should be on separate lines	25,674	(3.7%)	.73
7	Control flow statements “if”, “for”, “while”, “switch” and “try” should not be nested too deeply	24,513	(3.5%)	.34
8	Sections of code should not be “commented out”	22,039	(3.2%)	.52
9	Source files should not have any duplicated blocks	16,456	(2.4%)	.22
10	“@Override” should be used on overriding and implementing methods	16,291	(2.4%)	.64

RQ₄: What are the most costly to fix types of code technical debt in the studied ecosystem?

To answer this question we summed up all the open issues from all the analyzed systems but this time, we looked at the effort instead of the frequency. We are still only looking at the issues that are still open in the last analyzed commit.

Table 2.5 shows the ten most expensive in terms of effort types of issues in the analyzed projects. By analyzing the information in the table, we observe that:

- Code duplication, is the most expensive to fix in terms of the estimated required time. The function for estimating the time required to remove duplication estimates the effort linearly with the cardinality of the clone.
- Just as with the frequency, exception handling is again the most time-consuming problem to fix. The two types of issues regarding exceptions, account together for more than 13% of the estimated time for paying back the technical debt.
- Rule 3, is responsible in the ecosystem for 8.4% of the effort. Compared with the Apache Sling system presented in the Motivating Example section which had 13%

Table 2.5: The ten most costly to fix types of technical debt in the Apache ecosystem

#	Issue	Effort in minutes		Gini
1	Source files should not have any duplicated blocks	967,490	(13.8%)	.33
2	String literals should not be duplicated	642,122	(9.2%)	.36
3	Generic exceptions should never be thrown	587,860	(8.4%)	.47
4	Cognitive Complexity of methods should not be too high	353,527	(5.0%)	.37
5	Exception handlers should preserve the original exceptions	334,670	(4.8%)	.39
6	Methods should not be too complex	257,213	(3.7%)	.34
7	Control flow statements “if”, “for”, “while”, “switch” and “try” should not be nested too deeply	245,130	(3.5%)	.34
8	The members of an interface declaration or class should appear in a pre-defined order	193,780	(2.8%)	.43
9	Dead stores should be removed	165,990	(2.4%)	.42
10	Standard outputs should not be used directly to log anything	154,390	(2.2%)	.52

this is much lower. This would probably be useful information for the developers of Sling.

Since we cannot present all the 232 issues uniquely detected by the tool, we summarize their magnitude by computing again the Gini coefficient for the estimated effort per issue. Summing up the percentage of all the issues in Table 2.5 shows that 55.8% of all the estimated effort is due to these ten issues. We conjecture that if progress was made towards eradicating some of top problematic issues, the community would make considerable progress in avoiding much technical debt.

2.5 Threats To Validity

In this section, we present and discuss possible threats to the validity of our study.

Construct validity reflects how far the studied phenomenon is connected to the intended studied objectives. The main threats related to construct validity are due to possible inaccuracy in the identification of technical debt. Since we relied on the default SonarQube rules and the default threshold for each rule in order to detect the violations leading to technical

debt, the results are subject to the Software Quality Assessment based on Lifecycle Expectations model assumptions. This threat is partially mitigated by the fact that the analysis of technical debt evolution implies a relative rather than an absolute assessment of technical debt for the examined systems.

Since the Research Questions have been investigated through a case study, threats to the reliability should be examined. Reliability is linked to whether the experiment is conducted and presented in such a way that others can replicate it with the same results. We believe that the documentation of the adopted research process along with the online replication kit will facilitate any researcher who is interested in replicating this study.

Finally, as in any case study, external validity threats apply, limiting the ability to generalize the findings derived from sample to the entire population. However, the sixty-six systems that we analyzed have been randomly selected from the Apache ecosystem, and represent above a quarter and below a third of all contained Java projects. Moreover, we do not claim that the results on TD evolution or the types of TD hold for other Java systems or different ecosystems.

2.6 Related Work

This section reports the studies that are related to our work. Specifically, we report empirical studies that study the Apache ecosystem, studies that deal with the introduction, evolution, and the survivability of the code smells on OSS projects and finally, studies that study the impact of code smells on the OSS projects.

Evolving code smells and software metrics The evolution of code smells has been studied extensively. One of the first studies is by Olbrich et al. (Olbrich et al. 2009), who investigated the evolution of two code smells, namely God Class and Shotgun Surgery. They analyzed historical data of two large OSS projects from the Apache foundation: Apache Lucene and Apache Xerces 2 J, the results of their study report (i) that during the evolution of the projects there are phases that the number of these code smells decreases and phases that this number increases and (ii) the size of the system does not affect these changes.

Zazworka et al. (Zazworka et al. 2011) conducted a case study on the design debt. They analyzed two sample applications by a software development company and they investigated how God Classes affect the maintainability and the correctness of the projects. The results of their study show that God Classes have higher change-proneness when they are compared to the non-God Classes. Furthermore, they suggest that God Classes be seen as instances of technical debt and also they point that if the developers split the God Classes into multiple smaller classes that could lead to the generation of more problematic classes and that would have as result an increment to the number of the files that has to be edited.

Peters and Zaidman (Peters and Zaidman 2012) also conducted a case study on the lifespan of the following code smells: God Class, Feature Envy, Data Class, Message Chain Class, and Long Parameter List Class. They mined open-source projects and their main finding reports that the engineers are aware of the existence of the code smells in their systems but they do not worry for their impact and that has as result to perform very few refactoring activities. The main difference between our study and their that they analyzed only a small number of Java projects (only seven) and they focused their study only on five code smells. Furthermore, they did not measure how much effort is required in order to remove them.

Chatzigeorgiou and Manakos conducted a study on the evolution of code smells and they report that as the projects evolve over time the number of code smells increases (Chatzigeorgiou and Manakos 2014). Furthermore, the developers of the projects perform very few actions in order to remove the code smells from the projects.

All the previous studies focused on a limited number of types of smells and small number of systems. In contrast, Curtis et al. (Curtis et al. 2012) performed a large-scale study on many business applications. They used more than 1200 rules of good architectural and coding practice and they reported the TD of 745 business applications. The main difference between their study and ours is the focus: we focused only on Java open-source software projects by the Apache Foundation, they analyzed a big number of business applications that have been developed on many languages as diverse as COBOL, C++, .NET, ABAP, and Java. The similarity between their study and ours is that we also used a set of good architectural and coding practices

Software Ecosystems. Software ecosystems have been studied in many contexts: their evolving size and developer activity (Vasilescu et al. 2014, Lungu et al. 2009), their evolving dependencies (Bavota et al. 2013, Decan et al. 2016), their API evolution (Robbes et al. 2012). The very ecosystem that we study in this paper, Apache, has been studied from the perspective of sentiment analysis on the mailing lists (Tourani et al. 2014) and the evolution of dependencies between the projects in an ecosystem (Bavota et al. 2013).

One study on open-source systems that comes close to ours in its focus is the one of Tufano et al. (Tufano et al. 2015) who conducted an empirical study on 200 open-source software projects. They analyzed projects from three ecosystems namely Apache, Android, and Eclipse and they investigated questions about code smell life cycle. They found that the most code smells are introduced with the creation of the class or file when the developers implement new features or enhance already exist ones. They also report that the majority of the smells are not removed during the project's evolution and few are removed as a direct consequence of refactoring operations. Our study differs from their work in that we focus on the trends at the system level, and we also consider the estimated time that is required in order to resolve the issues of a project

2.7 Conclusions and Future Work

In this paper we have studied sixty-six Java systems from the Apache ecosystem. We analyzed cumulatively more than 16,000 weekly commits and we mined 695,731 project issues as reported by SonarQube. From this data we have learned that in the majority of the systems that we studied, there is a significant increase trend on the size, number of issues, and on the complexity metrics of the project. On the other hand, the normalized technical debt decreases as the project evolves.

Some of the most frequently occurring issues regard low-level coding problems some of which could probably be decreased with good IDE support (e.g. duplicated strings). On the other hand, the most expensive types of technical debt that must be paid back in the ecosystem are actually higher-level problems: duplicated code and ad-hoc exception handling. Exception mis-handling is more unevenly distributed in the ecosystem than code duplication.

One of the reasons for which this study did not analyze the entire Apache ecosystem but rather a sample of it is the slowness of the analysis using SonarQube for which the computation time required is linear with the number of versions. In order to allow the analysis of more systems and also a finer temporal granularity level than a week, in the future we will investigate approaches that would provide better scalability.

In the paper we also observed that a very small minority of problem types is responsible for the vast majority of estimated technical debt. We conjectured that if progress was possible towards *preventing* some of the top problematic issues the community could *avoid incurring* a large percentage of the technical debt in the first place. Even if for other communities the problem ranking would be different, we believe that the approach of aggregating the information from system level to the entire ecosystem will always provide valuable insights. Indeed we consider this to be one of the take-away messages of this study.

Finally, although larger than many earlier studies on the evolution of technical debt in open-source systems, our study is still limited to a random sample from one ecosystem. It would be valuable if these results would be replicated by other researchers in other open-source ecosystems, and maybe also in other languages.

Published as:

G. Digkas, M. Lungu, P. Avgeriou, A. Chatzigeorgiou and A. Ampatzoglou, “How do developers fix issues and pay back technical debt in the Apache ecosystem?,” *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Campobasso, 2018, pp. 153-163, doi: 10.1109/SANER.2018.8330205.

Chapter 3

How do developers fix issues and pay back technical debt in the Apache ecosystem?

Abstract

During software evolution technical debt (TD) follows a constant ebb and flow, being incurred and paid back, sometimes in the same day and sometimes ten years later. There have been several studies in the literature investigating how technical debt in source code accumulates during time and the consequences of this accumulation for software maintenance. However, to the best of our knowledge there are no large scale studies that focus on the types of issues that are fixed and the amount of TD that is paid back during software evolution. In this paper we present the results of a case study, in which we analyzed the evolution of fifty-seven Java open-source software projects by the Apache Software Foundation at the temporal granularity level of weekly snapshots. In particular, we focus on the amount of technical debt that is paid back and the types of issues that are fixed. The findings reveal that a small subset of all issue types is responsible for the largest percentage of TD repayment and thus, targeting particular violations the development team can achieve higher benefits.

3.1 Introduction

Technical Debt (TD) is a powerful metaphor that represents shortcuts taken in a software development project, usually to meet business goals such as limited time or budget (Kruchten et al. 2012). Technical debt concerns mostly invisible parts of the system (i.e. no visible features or defects) that have a detrimental effect on the maintainability and evolvability of software (Kruchten et al. 2012). Technical Debt cannot realistically be eliminated but it does need to be proactively managed in order to be kept at a sustainable level (Li et al. 2015).

One of the primary activities of Technical Debt Management is termed Technical Debt Monitoring (Li et al. 2015): the activity of continuously inspecting the levels of technical debt throughout time. Monitoring technical debt aims to identify trends in the evolution of technical debt and alert the development teams in case certain technical debt items increase

beyond a threshold, or even worse grow out of control. In that sense monitoring helps to identify and prioritize repayment actions where technical debt items are resolved (e.g. through refactoring).

In this paper we study the technical debt evolution of fifty-seven open-source software projects by the Apache ecosystem¹², through a longitudinal case study, with the aim of identifying cases in which repayment activities take place. Although in the literature there are many studies which investigate projects by the Apache Software Foundation (since they are highly appreciated and used by many software engineers), in this work we narrow down our scope to the types of issues that are fixed and the amount of technical debt that is paid back over time.

Structure of the Paper The rest of the paper is organized as follows: In Section 3.2 we discuss related work on the evolution of software projects with respect to their quality and code smells in particular. Section 3.3 clarifies methodological issues related to the collection of data for the repayment of technical debt. The protocol of the case study is outlined in Section 3.4, while the Results are presented and discussed in Section 3.5. Key observations and open questions resulting from the study are outlined in Section 3.6. Threats to the validity of the study are discussed in Section 3.7. Section 3.8 provides information regarding the replication package for the study. Finally, we conclude in Section 3.9.

3.2 Related Work

This section reports studies that are related to our work. These include empirical studies that investigate the evolution of open-source software projects, the evolution of technical debt over time and studies that deal with the survivability and impact of code smells in open-source software projects.

Bavota et al. (Bavota et al. 2013) have studied the evolution of inter-dependencies of 147 Java projects over a period of 14 years that belong in the Apache ecosystem. The results of their study show that there is a linear increase trend in the number of projects but the number of dependencies between them tend to increase exponentially. They have also found that the developers do not upgrade immediately the dependencies of their projects unless it is a major release.

One of the first studies that investigates the evolution of code smells in the literature is by Olbrich et al. (Olbrich et al. 2009). They analyze historical data of two projects by the Apache Foundation, namely Apache Lucene and Apache Xerces 2 J and they study how the

¹Apache Software Foundation is one of the biggest communities which “provide software products for the public good”

²In this paper we consider an ecosystem to be “a collection of software systems that are developed and co-evolve in a shared environment”(Lungu 2009) even if other definitions exist (Manikas 2016)

God Class and Shotgun Surgery smells evolve over time. The main findings of their study are that during the evolution of these projects, there are phases where the number of these smells decreases and phases where the number increases. Moreover, these phases are not affected by the size of the systems.

Peters and Zaidman (Peters and Zaidman 2012) developed a tool which is called SAC-SEA. The tool computes the lifespans of the following code smells: God Class, Feature Envy, Data Class, Message Chain Class, and Long Parameter List. The tool has been employed to analyze eight open-source software projects. The main finding of their study reports that software engineers are aware of the existence of the code smells in their projects. However, there is evidence that they perform very few refactoring activities.

Chatzigeorgiou and Manakos (Chatzigeorgiou and Manakos 2014) have also investigated the evolution of code smells in object-oriented projects. They studied the evolution of Long Method, Feature Envy, State Checking, and God Class smells throughout successive versions of two open-source software projects, namely: JFlex and JFreeChart. The results of their study show that as the projects evolve over time the number of code smells increases. Furthermore, the developers do not perform refactorings in order to solve these code smells and in the vast majority of the cases if one code smell is removed, it is probably a side effect of regular maintenance and not a result of intentional refactoring activity.

Zazworka et al. (Zazworka et al. 2011) investigate how God Classes impact maintainability and correctness. To this end, they analyzed two sample applications by a software development company. Their findings show that God Classes are more change-prone and there are some cases where they also are more defect-prone, when they are compared to the non-God Classes.

Tufano et al. (Tufano et al. 2015) conducted a study on 200 open-source software projects from three different software ecosystems, namely: Apache, Eclipse, and Android. The main aim of their study is to understand when and why code smells are introduced into the projects and what is their life cycle. In order to answer this question, they focus on the following code smells: Blob Class, Class Data Should be Private, Complex Class, Functional Decomposition, Spaghetti Code. The results of their study, show that a) in the majority of the cases, the code smells are introduced into the projects with the creation of classes or files, b) also, while the project evolves over time, the smelly code artifacts will become more smelly, c) software engineers introduce new code smells when they implement new features or when extend the ones that already exist, d) the developers who introduce new code smells into the projects, are the ones who work under pressure and not necessarily the newcomers, and e) the majority of the smells are not removed during the project's evolution and few of them are removed as a direct consequence of refactoring operations.

Digkas et al. (Digkas et al. 2017) have also studied the evolution of open-source software projects by the Apache Foundation. They analyze the evolution of 66 Java projects, over a period of 5 years, using SonarQube in order to investigate how technical debt evolves and

Table 3.1: Related Works

Authors	Year	Projects	Issues
(Bavota et al. 2013)	2013	147	-
(Zazworka et al. 2011)	2011	2	1
(Olbrich et al. 2009)	2009	2	2
(Chatzigeorgiou and Manakos 2014)	2014	2	4
(Peters and Zaidman 2012)	2012	8	5
(Tufano et al. 2015)	2015	200	5
(Digkas et al. 2017)	2017	66	232
(Maldonado et al. 2017)	2017	5	-
(Curtis et al. 2012)	2012	745	> 1200

what are the types of Technical Debt issues that incur it. In their study, they use the default SonarQube settings and they examine 232 different types of Technical Debt issues that incur technical debt. The results of their study show that on the one hand, there is a significant increasing trend on the size, complexity, number of Technical Debt issues, and the technical debt over time. But on the other hand, when the technical debt is normalized to the size of the project, it decreases over time. This means that the developers either insert better quality code on the projects or they fix some of the open Technical Debt issues.

A large-scale study on 745 business applications has been performed by Curtis et al. (Curtis et al. 2012). The projects that they analyzed are from 160 companies that belong in 10 industry segments. The analysis that they perform is similar to that of the present study. They used more than 1200 rules of good architectural and coding practices in order to evaluate the quality of these applications. One key difference between our and their study is that we investigate the number of the issues that are fixed and the amount of technical debt that is paid back.

Maldonado et al. (Maldonado et al. 2017) have conducted an empirical study on the removal of Self-Admitted Technical Debt (SATD) by analyzing the code comments of five open-source software by the Apache Software Foundation. The results of their study show that the removal of SATD issues could take from several months and up to a decade. The authors also found that in most of the cases the developer who introduced an SATD issue is the one who removes it when they fix bugs or add new features.

Table 3.1 summarizes all studies reported in this section, including the year of publication, the number of analyzed projects and the number of examined issues.

3.3 Studying Technical Debt Evolution

In order to study the repayment of debt one must study its evolution, and in order to study the evolution of debt one must be able to detect it in the first place. Different types of debt can be detected from various sources of information: source code, architectural documentation, issue trackers, mailing lists, etc. However, to detect the evolution of debt over a long period of time and in a large number of systems, restricting the study of debt to that which can be inferred based on the study of source code makes the challenge more approachable.

3.3.1 Using SonarQube for Debt Estimation

In this study, we use SonarQube (Campbell and Papapetrou 2013) to detect fixes of rule violations and in this way we compute the amount of technical debt that is paid back. SonarQube, relies on a set of rules in order to calculate the amount of technical debt. During the analysis of any project, SonarQube creates a new *issue* every time a piece of code breaks one of the predefined *rules*. For every issue it assigns an estimate of how much time will be required for someone to solve it. A downside of this decision is the restriction of the study to the particular types of technical debt that can be detected by source code analysis and in particular by the SonarQube tool platform.

We have decided to use SonarQube because it is one of the most popular tools that are used in industry for the purpose of measuring and estimating debt. Moreover, SonarQube can detect a large number of source code related types of technical debt.

The tool uses different strategies for estimating technical debt for different types of issues:

1. **Constant Time** – some issue types are assigned a constant time estimate for their fixing (e.g. “Sections of code should not be commented out” is assigned a constant fixing time independent of the size of the commented out code)
2. **Issue Dependent Time** – some types of issues are estimated to take a time which takes into account their particular characteristics (e.g. cloning takes into account the number of clones)

Finally, to compute the total amount of technical debt per system at a given time, SonarQube calculates the estimated fixing time for all the open issues.

3.3.2 Studying Multiple Revision of a System

SonarQube, and other similar tools, cannot take as input *deltas* between system revisions, but instead they parse the entire source code of a system and build a model for every new version submitted for analysis. This means that even if a single line change in a single file

will result in a massive and time-consuming analysis step which for large systems can take several minutes.

To study the evolution of a large number of systems one must thus *discretize* their evolution by analyzing only a limited number of revisions of the system. However, although the analysis of multiple revisions is slow, the individual *issues* together with their cost estimation being the most basic unit of debt are identified by the tool across revisions and their identity can be preserved.

The preservation of issue identity is achieved even in the presence of different types of changes. However, one of the situations in which the identity of an issue is lost is whenever a file is renamed and at the same time the content of that file is changed too. On the other hand, if only one of these changes occurs (i.e. renaming of the file or change of content) the tool is still able to preserve the identity of the issues.

3.3.3 Detecting Fixed Issues

When analyzing multiple revisions of a system, SonarQube tracks the issues that are fixed and the debt that is repaid. When an issue disappears, it is considered **fixed**.

By investigating manually a series of such fixed issues we observed three types of situations in which the tool considers an issue to be fixed:

1. Source code deletion. If a piece of code that contains some issues that incur technical debt is deleted, then the technical debt is paid back automatically. Deleting a piece of code (e.g. a statement of a block) represents technical debt repayment which can be either intentional or unintentional.
2. Refactoring of the code aiming explicitly at the removal of the issues. If one section of code has been detected to contain technical debt issues, the refactoring of the corresponding code to remove the issue will result in the repayment of TD.
3. A file-rename-con-file-content-change situation, as described earlier. In this problematic situation, the tool considers all the issues in the old file to be removed and a series of equivalent issues in the new file to have appeared.

Since the goal of this study is the detection of fixed issues and thus, repaid debt, we must be wary of the issues which are fixed in the third situation since they are not actually *fixed* and the corresponding debt is not being *repaid*.

By using the SonarQube API a filter can be implemented to remove the issues in the third category from the list of fixed issues³. By using such a filter, in the remainder of this paper, we only talk about the first two types of closed issues.

³Code for the filter is available in the online replication package for the paper

3.3.4 Classifying Issues

SonarQube classifies the issue types into two main categories: based on their type and severity.

Issue Types If an issue is related to a piece of code that is demonstrably wrong is classified as **bug**. If a piece of code could be exploited by a third person and that could be the reason to harm the system then it is classified as **vulnerability** and finally, as **code smells** are classified all the issues that represent an instance of improper code and they are neither a bug nor a vulnerability.

Issue Severity Levels In terms of severity, there are five categories namely: blocker, critical, major, minor, and info. Based on the main object of their impact, there are the following three sub-categories:

1. *Impact on the system.* **Blocker** and **critical** designation expresses the impact that an issue can have in the behavior of the application in production. Blocker issues have high probability to negatively impact the system, which is a reason that the software engineers should fix them as soon as possible. Critical issues have lower probability, when we compare to the blocker, to impact negatively the system.
2. *Impact on the productivity of a software engineer.* **Major** can have high impact and **minor** have little impact.
3. *Everything else.* The last category is the **info**. This category collects all the issues that are neither a bug nor a quality flaw.

3.4 Study Design

The goal of our study is to analyze the evolution of open-source software projects in the Apache ecosystem for the purpose of understanding and investigating the types of the issues that have been fixed and the amount of technical debt that is paid back. More specifically, our study aims at addressing the following five research questions (RQs):

1. *How does the code TD issue fixing rate vary for different projects?* The goal of this research question is to investigate whether software development practices differ with respect to technical debt repayment among projects.
2. *What is the fixing prevalence of the various code TD issue types?* This RQ aims at identifying which issues are fixed more often than others.

3. *How does the fixing rate vary for different code TD issue types?* Because the number of fixes is dependent on the number of issues for each particular type, this RQ investigates the fixing rate, i.e. the percentage of fixes over the total number issues present in each system.
4. *How is the effort of paying back code TD distributed across the various types?* Since different issue types demand different amounts of effort to resolve them, this RQ sheds light into the issues which have yielded the higher benefit in terms of technical debt repayment.
5. *After how much time is code TD paid back?* Similarly to previous studies we analyze the time that is required to fix issues, for the types that have been identified as more beneficial in terms of technical debt repayment.

3.4.1 Project Selection

For the purpose of this study, we chose to analyze projects by the Apache Software Foundation. Since the analysis that we performed is computationally intensive, we used the Apache Software Foundation Index⁴ in order to randomly select a sample of fifty-seven Java projects.

The three main inclusion criteria that we used are the following:

1. In terms of programming languages, we chose to analyze projects that the **main programming language is Java and have at least 100 classes**
2. In terms of project evolution, we included projects that have been developed for **at least two years and 1000 commits**, and
3. In terms of activity, we included only projects that there was commit activity in 2017, in other words, projects that are **still active in the year when we conducted this study**

Figure 3.1 shows considerable variation in both the number of commits and the size of the systems as measured by counting the number of Java files.

3.4.2 Issue Selection

We are limiting the focus of this paper to those issues in the blocker-critical-major category: bugs or flaws that have a probability of affecting the behavior of the application or can highly impact the productivity of developers. The reason for this is that many of the minor and info issues are quite *minor* including:

⁴<https://projects.apache.org/projects.html?language#Java>

Figure 3.1: Correlation between the size of a project (in number of files) and commit density

- “Functions should not be defined with a variable number of arguments” (info)
- “Field names should comply with a naming convention” (minor)
- “Useless imports should be removed” (minor)
- “Unused local variables should be removed” (minor)

Figure 3.2 illustrates the results of aggregating the issues in the analyzed system based on their severity level. In total 5,041 Blocker, 44,421 Critical, and 105,847 Major issues have been fixed in the full history of the fifty-seven systems under study. The light gray background presents the total number of issues of that kind. The Major have a much higher percentage of fixes than the other types (see Figure 3.2).

This narrowing of the focus results in 160 types of issues to which we will limit our discussion to henceforth.

Figure 3.2: Distribution of resolved issues by priority type (the light gray background presents the total number of issues of that kind)

Figure 3.3: Number of open (light yellow) and closed (dark green) issues per project. The top part of the figure presents the sizes of the corresponding projects; the middle absolute numbers of open/closed issues; the bottom part presents percentages of open/closed issues

3.5 Results and Discussion

3.5.1 How does the code TD issue fixing rate vary for different projects? (RQ₁)

The goal of this research question is to investigate whether software development practices differ with respect to TD repayment among projects.

Figure 3.3 shows several types of information for every project. The middle histogram presents the absolute number of issues fixed in a project (dark green) and the number of issues which are still open (light green). The bottom histogram presents in dark green the **issue fixing rate** – the percentage of the issues in that project which are closed⁵. The top part of the Figure 3.3 presents the size of the corresponding systems

Based on the figure we observe that:

- The highest fixing rate is presented by projects *nutch* and *jmeter* which both have more than 70% of the issues that appeared during their evolution fixed. This could be possibly attributed to the fact that the development team of *nutch*⁶ uses SonarQube to evaluate its quality, while *jmeter*⁷ is itself a SonarQube plugin.
- Four projects have a fixing rate of less than 10%. They are projects of varying sizes, with *polygene-java* having more than 2K Java files.
- If we compress further the information about fixing rates as in Figure 3.4, which shows a histogram depicting the number of projects for several classes of fixing rate with a bin size of 10%, the largest category, with 20 elements, corresponds to a fixing rate of between 20% and 30%.

Figure 3.4: Distribution of fixing rates in examined projects for bins of size 0.10

- There is no clear correlation between the absolute number of issues found in the system and the percentage of fixes: *batik* (rightmost) and *nifi* (7th from right) have very

⁵The projects are sorted in decreasing order of their issue fixing rate

⁶<https://svn.apache.org/repos/asf/nutch/site/publish/sonar.html>

⁷<https://docs.sonarqube.org/display/SONARQUBE45/JMeter+Plugin>

large numbers of issues but very low fixing percentages while for similar numbers of issues `nutch` (leftmost) and `jmeter` (2nd from left) have very high fixing rates.

- There seems to be no clear correlation between the size of the projects and the fixing rate: `sis` (4th from left) with one of the highest fixing ratios and `batik` (rightmost) with the lowest fixing ratio are about the same size.

3.5.2 What is the fixing prevalence of the various code TD issue types? (RQ₂)

To answer RQ₂ we sum up all fixed issues, without differentiating among projects. The histogram of Figure 3.5 presents the distribution of the number of fixed issues for each type: every vertical bar is an issue type, and its height is proportional to the number of issues of that type. The figure shows a strongly skewed distribution where a handful of issue types are fixed very frequently while for most types fixes are rarely encountered.

Computing the Gini index for the prevalence of the issues we obtain a very high value of 0.845⁸. This means that the “wealth” of issue fixes is very inequally distributed across the issue types.

Figure 3.5: Distribution of resolved issues by issue type for the 160 types of issues in this study

In order to focus on the left side of Figure 3.5, Table 3.2 shows the ten most frequently

⁸The index varies between 0 for a population with perfect equality and 1 for a population with perfect inequality

fixed types of issues encountered in the analyzed projects. Instead of presenting the absolute number of fixed issues for each issue type, we present the prevalence (column P%) which is, the percentage of that type of issues fixed from the total number of fixes.

Table 3.2: Most frequent types of resolved code smells (Types related to exceptions, complexity, and duplication are highlighted)

Issue	P% ^a
1. Statements should be on separate lines	7.8
2. Sections of code should not be "commented out"	7.8
3. String literals should not be duplicated	6.4
4. Exception handlers must preserve the original exceptions	6.1
5. Control flow statements should not be nested too deeply	5.9
6. Cognitive Complexity of methods should not be too high	5.4
7. Generic exceptions should never be thrown	5.1
8. Standard outputs should not be used directly for logging	3.9
9. Source files should not have any duplicated blocks	3.8
10. Synchronized classes Vector, [...] should not be used	2.8

^a Prevalence percentage

Several high-level observations that can be derived from the table are:

1. **Skewdness.** The ten issue types (out of 160!) in the table account for 55.14% of all the issue fixes in the studied systems. All the fifty-seven systems contain fixes of these issues with the exception of issue #1 missing in five systems and issue #10 absent from 2 systems.
2. **Ease of fix.** The most frequently occurring fixes are also probably some of the easiest to be remedied, as separating statements to different lines (issue #1), removing commented code (issue #2), and removing string literal duplication (issue #3) that normally have no side effects.
3. **Duplication.** Issues related to duplication appear twice in the table (#3 and #9), are found in all the analyzed systems, and account together for more than 10% of the total issues fixed revealing that code clones are identified and handled by developers in numerous instances.
4. **Exception Handling.** Issues related to exception handling also appear twice in the table amounting together for more than ten percent of the total fixes. They also affect all the systems.

Types of Issues It should be noted that most of the fixed issues are not necessarily language specific; the first language specific issue is at position 10, namely “Synchronized classes Vector, Hashtable, Stack and StringBuffer should not be used”.

All the issues in the table are classified by SonarQube as code smells, i.e symptoms that possibly indicate a deeper problem, but are not bugs. The ten most frequently fixed problems that are classified as bugs are shown in Table 3.3.

Table 3.3: Most frequent types of resolved bugs and vulnerabilities

Issue	P% ^a
Bugs	
16. Resources should be closed	2.15
22. Null pointers should not be dereferenced	1.30
36. InterruptedException should not be ignored	0.49
41. Floating points should not be tested for equality	0.43
45. Conditionally executed blocks should be reachable	0.38
70. All branches in a conditional structure should not have exactly the same implementation	0.11
72. Identical expressions on 2 sides of binary operator	0.10
78. Assignments should not be redundant	0.08
79. Jumps should not occur in “finally” blocks	0.08
80. Classes should not be compared by name	0.08
Vulnerabilities	
68. SQL binding mechanisms should be used	0.11
76. Credentials should not be hard-coded	0.08
127. File.createTempFile should not be used to create a directory	0.01

^a Prevalence percentage

Finally, with respect to issues tagged as vulnerabilities, there are only three vulnerability types, in the blocker-critical-major category, that have been detected as fixed. These fixes are responsible for less than 0.20% of the total fixes, as shown in Table 3.3. Although not shown in the table, these issues occur in very few projects, and also have a low fixing rate.

3.5.3 How does the fixing rate vary for different code TD issue types? (RQ₃)

The reason for asking this research question is understanding whether some types of technical debt are repaid more often than others – and thus, could represent candidates for debt

Figure 3.6: Fixing rate (horizontal axis) vs. Number of Issues for a given Issue Type

that is more important for the developers of the Apache ecosystem⁹.

By computing the fixing rate as the ratio between all the issues of a given type that are closed over the ones which are open we found six issue types with a fixing rate of 100%. However, they are very rare with the most popular one (“Future keywords should not be used as names”) occurring only 21 times and the second most popular one (“Variables should not be self-assigned”) occurring only 10 times in the whole the history of all the fifty-seven projects¹⁰. The reduced prevalence of the issues decreases their relevance for a broader developer community. To increase the relevance of the observations, in the remainder of this section, we limit our discussion to issues that have a minimum of 500 discovered occurrences.

Issues with at least 500 occurrences Figure 3.6 shows a scatter plot of the 69 issue types which have at least 500 occurrences¹¹. The red data points correspond to the ten issue types with the highest fixing rate which are detailed below. The figure shows that:

- The majority of the issue types have a fixing rate between 20% and 50%

⁹There is no guarantee that if something has a high fixing rate it is necessarily important for the developers; however, we expect that those issues which are important, will be found among those with a high fixing rate. On the other hand, it is safe to say that issue types with low fixing rate, are very likely to not be important for developers

¹⁰The others are available in the online replication package

¹¹By removing the other issue types we are removing a lot of “noise” data points that are crammed on the bottom of the figure along the axes

- The issue types with the highest fix rates (marked in red) are distinct from the issue types with the highest prevalence in the studied systems.

Table 3.4 presents details about the ten issue types with the highest fixing rate. There are eight issue types which are fixed in at least 50% of the cases. One salient observation is that, in the table, the majority of the issues are Java specific (marked with ☕).

Table 3.4: Issues with the highest fixing rate (☕are Java-specific)

Issue	Count	F ^a
Conditionally executed blocks should be reachable	594	59
☕ Replace Map.get/test with single method call	694	58
☕ Deprecated elements should have both the annotation and the Javadoc tag	2678	57
Unused "private" fields should be removed	2094	56
Boolean expressions should not be gratuitous	1571	55
☕ Synchronized classes Vector, Hashtable, Stack and StringBuffer should not be used	4381	53
☕ Constructors should not be used to instantiate "String" and primitive-wrapper classes	3761	52
Dead stores should be removed	3720	50
☕ @Override should be used on overriding [...]	1586	48
Unused "private" methods should be removed	689	47

^aFixing Rate

Table 3.5 lists the ten issue types with the lowest fixing rate. The table contains fewer Java-specific issues (it could be argued whether there is one or several). Instead it shows generic issues related to best practices which were seen in an earlier discussion to have a high prevalence in the case study: exception handling, complexity, duplication.

3.5.4 How is the effort of paying back code TD distributed across the various types? (RQ₄)

This question aims to shed light into the benefits resulting from paying back technical debt, as the reduction of the technical debt principal is not only related to the frequency of fixes but also the relative contribution of each issue to the overall technical debt.

To answer this research question we have summed up the effort that would have been nominally required to resolve all fixed issues, according to the estimates provided for each issue type by SonarQube. The total amount of technical debt estimated to be paid back is

Table 3.5: Issue types with the lowest fixing rate

Issue	Count	F ^a
Floating point numbers must not be tested for equality	665	28
Expressions should not be too complex	597	26
Exception handlers should preserve the original exceptions	9535	26
Methods should not be empty	3549	26
☞ Lambdas and anonymous classes should not have too many lines of code	869	25
Instance methods should not write to "static" fields	547	25
Generic exceptions should never be thrown	7961	23
Resources should be closed	3338	22
Utility classes should not have public constructors	746	22
String literals should not be duplicated	9875	19

^aFixing Rate (rounded to the closest integer)

30,200 hours and Table 3.6 shows the percentage of repaid technical debt by issue type. The "Change in ranking" column from the table shows the relative change in the global ranking of a given issue with respect to the prevalence ranking discussed in RQ_2 .

Changes in Ranking By analyzing the "Change in ranking" column we observe that the order of issues changes with respect to the frequency of fixes presented in RQ_2 . This means that the most frequently fixed issues are not necessarily the ones representing the highest amount of technical debt paid back. The biggest jump in the ranking is observed for the issue of duplication between source files. The fixes of this type of issue are responsible for $\approx 12\%$ of the total technical debt that is paid back. That is because the estimation of technical debt payback time is proportional to the number of duplicated blocks.

Types of Debt The table shows that three general classes of debt are represented by more than one issue in the top-10 list and together account for more than half the total repaid debt:

1. **Method Complexity** – accounting for about 24% of the total estimated pay back effort – is detected in three cases: issue types #2, #6, and #8. These issues are related to the understandability of the code. Cognitive Complexity measures the maintainability of the code and indicates how difficult is for someone to understand the control flow of a method and to maintain it. Deep nesting of control flow also increases cyclomatic complexity and reduces understandability and testability of the code. Having a code base clear without commented out sections of code also helps software engineers to

Table 3.6: Smells whose resolution has yielded the higher benefit

#	Issue	Est. Repaid Debt	Change in Ranking ^a
1	Source files should not have any duplicated blocks	11.78	↑8
2	Cognitive Complexity of methods should not be too high	11.19	↑4
3	Generic exceptions should never be thrown	8.83	↑4
4	String literals should not be duplicated	6.64	↓1
5	Exception handlers should preserve the original exceptions	5.29	↓1
6	Control flow statements "if", "for", "while", "switch" and "try" should not be nested too deeply	5.07	↓1
7	Synchronized classes Vector, Hashtable, Stack and StringBuffer should not be used	4.86	↑3
8	Methods should not be too complex	3.97	↑3
9	Standard outputs should not be used directly to log anything	3.37	↓1
10	Sections of code should not be "commented out"	3.36	↓8

^arefers to changes compared to the frequency of code smells.

understand the functionality easier. Commented out sections of code slow down the development and maintenance speed.

2. **Code Duplication** – accounting for about $\approx 18\%$ of the total estimated pay back effort – is detected in two situations: at file level and at the level of string literals. Code duplicates appear frequently and cost a lot in many different ways. They make the source code difficult to understand, maintain and extend.
3. **Exception Handling** – accounting for about 14% of the total estimated pay back effort – is represented by two cases: issue types #3 and #5. Using generic exceptions prevent methods from handling more specific, application-generated errors in the appropriate way. Moreover, the results imply that developers care about passing forward the original exceptions' messages and stack traces.

Table 3.7: Bugs and Vulnerabilities whose resolution has yielded the higher benefit

Issue	ERD ^a
Bugs	
17. Null pointers should not be dereferenced	1.12
23. Resources should be closed	0.93
31. InterruptedException should not be ignored	0.64
37. Conditionally executed blocks should be reachable	0.49
55. Jump statements should not occur in "finally" blocks	0.20
58. Floating point numbers should not be tested for equality	0.18
64. All branches in a conditional structure should not have exactly the same implementation	0.14
77. Blocks should be synchronized on "private final" fields	0.08
79. Non-thread-safe fields should not be static	0.08
83. Synchronization should not be based on Strings or boxed primitives	0.04
Vulnerabilities	
53. Credentials should not be hard-coded	0.211
56. SQL binding mechanisms should be used	0.195
134. File.createTempFile should not be used to create a directory	0.002

^aEstimated Repaid Debt **Figure 3.7:** Histogram of the survival time for the 155K issues in this study with bins of 30 days

3.5.5 After how much time is code TD paid back? (RQ₅)

This RQ focuses on the survival of the various types of issues that have been identified in the previous research questions focusing on the issues whose resolution has yielded the higher benefit in terms of TD payback. For each one of those issues types, we calculated the number of days that the issue survived, i.e. the difference between the time point at which an issue is introduced to the source code to the time it is resolved¹².

Figure 3.7 presents the histogram of the fixing time of all the 150K issues found fixed in this study. Taking into account the fact that the total number of fixed issues in this study is slightly more than 155K, the histogram shows that:

- Almost 20% ($\approx 30K/155K$) of the issues are fixed within one month of their introduction
- More than 50% of the issues are fixed within the first year
- The lifetime of the issues is a long-tailed distribution with a small number of issues being fixed even after ten years. It seems that one must never lose hope when it comes to the possibility of debt being paid back!

Figure 3.8 shows the distribution for each of the ten issue types reported in RQ₂ in the form of a boxplot (outliers are not shown in the diagram for simplicity). The figure limits itself to presenting only the respective issue types since, as shown in the previous section, these are responsible with more than half of the effort required for payback.

¹²One of the limitations of the weekly commits approach is that when we compute the time an issue “survived” we will consider that it was fixed in the last commit of the week even if it was actually fixed earlier that week. In fact, the worst case scenario is when the issue was fixed on the first day of the week, and we only consider it to be fixed in the last day of the week. Thus our results regarding the longevity of a given issue have a built-in maximum imprecision of seven days

Figure 3.8: Distribution of survival time for top the ten issues (according to RQ₂)

There is a large difference between the minimum and the maximum duration that is required in order to fix an issue.

The main observations from the survival times of these issues are similar to the corresponding observations about the entire populations. For all issue types in the figure:

- 25% of all instances disappear in less than two months
- 50% of all issue instances disappear approximately within one year from their introduction to the code
- The longest lived issue types exhibit a large variation that spans from 5 years to more than 10 years

Moreover, the figure highlights in similar colors issue types which are similar. One can see that there is a certain similarity in the survivability distribution of the similar issue types.

3.6 Observations and Open Questions

Based on this study, we can conclude with several observations about the way technical debt evolved in the Apache Ecosystem over ten years:

- There is a very large variation between the fixing rate of issues that contribute to technical debt among the projects of the same ecosystem
- Only a very small minority of the issue types have a fixing rate of more than 50%. If we consider those types with at least 500 instances, then the count is eight.
- There is a very large variation in the survivability of the different issues: about 10% of the issues are fixed within the first month, about 50% in the first year, and some of the issues can take up to ten years
- Issues related to duplication and exception handling are frequently encountered and rarely fixed by developers

Questions which are still open after this study:

- What percentage of the issue fixing happens during purposeful debt repayment by developers and what percentage happens as a side effect of source code evolution?
- What is the impact of individual developers on issue fixing and technical debt in the studied ecosystem?
- How can we use this information to be able to provide developers with ways of better understand their own system's evolution?

- Do other language ecosystems (Smalltalk, Javascript, R, have already been studied before (Decan et al. 2016, Robbes et al. 2012, Wittern et al. 2016)) have different attitudes towards debt repayment? How do these results compare against closed-source ecosystems?

3.7 Threats To Validity

The results of the study are unavoidably subject to external validity threats since a sample of projects has been analyzed to study how technical debt is paid back, thus limiting the ability to generalize the conclusions to projects of a different programming language, ecosystem or domain. Nevertheless, the fact that fifty-seven active, large and highly popular projects have been studied partially mitigates threats to generalization.

As explained in the study design, we consider treats an issue as paid back either when the development team intentionally refactored the code or when the code fragment hosting the issue (but not the entire file) has been removed. When the entire file had been deleted, we do not count the contained issues in that file in the removed ones. One threat to the construct validity of the study stems from that fact that issue elimination due to the removal of the corresponding statement, code block or method is consider as paying back technical debt whereas it could be a side effect of regular maintenance activities. However, even in the presence of this threat the findings on deliberate or unintentional technical debt payback can be valuable since the focus is on the relative frequency of each issue type rather than the absolute count of issue removals. An additional threat stems from the fact that technical debt identification is performed on source code artifacts rather than the built version of each project. The lack of build information often leads to false positives. For example, a language specific issue (e.g. the omission of the diamond operator) might be reported for a revision in which the rule does not apply (language version prior to Java 7).

Concerning threats to the repeatability, we believe that this study can be easily replicated by other researchers, since: a) the study protocol is extensively described in this paper, and b) the collection of data and analysis of the results did not rely on any subjective judgment and is based on SonarQube reports.

3.8 Data Set and Replication

As explained before, due to the limitations of multi-version analysis, collecting the data used for this study was time consuming and computationally intensive. In total, the analysis of 19,800 revisions of fifty-seven systems took more than a month even though we used multiple machines for the analysis. We took weekly snapshots for each one of the selected projects and we picked the last commit of each week. Then, for that revision we analyzed all the

Java files (including the test classes) in order to detect which of the open issues were fixed and which additional issues were introduced. The resulting PostgreSQL database dump is more than 28GB in size. An online Git repository available at <https://github.com/td-evolution-in-apache> contains a replication package which includes:

- A series of scripts that can be used for data analysis
- A spreadsheet with all the data that was used in this paper
- Information about how to obtain the full database dump and how to import it a SonarQube instance for those users who would want to explore it using SonarQube's API

We would be happy to support other researchers in using this dataset for their investigations as we believe there are more open questions that this study raises than it answers.

3.9 Conclusion and Future Work

The Technical Debt metaphor has been widely adopted and one of its underlying implications is that principal and interest are not regularly paid back. In analogy to bank clients who get into trouble when not paying their invoices promptly, increasing Technical Debt signifies software decay. However, development teams can act on Technical Debt by removing issues and inefficiencies.

In this paper we presented the results of a case study on fifty-seven Java projects from the Apache Software Foundation, focusing on the amount of technical debt that is paid back and the issues that are fixed. The results revealed that: a) a large percentage of technical debt is paid back during the evolution; however, for the majority of the projects the fixing rate is below 30%, b) a small percentage of issue types (such as complex methods, duplicates and exception handling problems) account for the majority of issue fixes and the majority of technical debt that is paid back and c) although some issue types live in the systems for several years, for the majority of the issues that get fixed, their elimination occurs within one year from their introduction. Such findings can be of help to development and maintenance teams by pinpointing the types of issues that have been adopted by other projects and yielded the largest benefit in terms of Technical Debt reduction.

Finally, although this study investigates a large number of open-source software projects and the issues that are fixed over time, it is still limited to a random sample of projects from only one ecosystem. Furthermore, it relies on SonarQube in order to detect the type of issues that are fixed and the amount of technical debt that is paid back. It will be valuable, in the future, to analyze a bigger number of projects, from more than one ecosystem, and projects that are developed in other languages or even multilingual projects. Moreover, we plan to replicate this study on all commits of the master branch including build information and performing further data sanitization.

Published as:

G. Digkas, A. Chatzigeorgiou, A. Ampatzoglou, and, P. Avgeriou, "Can Clean New Code reduce Technical Debt Density," *IEEE Transactions on Software Engineering*, doi: 10.1109/TSE.2020.3032557.

Chapter 4

Can Clean New Code reduce Technical Debt Density

Abstract

While technical debt grows in absolute numbers as software systems evolve over time, the density of technical debt (technical debt divided by lines of code) is reduced in some cases. This can be explained by either the application of refactorings or the development of new artifacts with limited Technical Debt. In this paper we explore the second explanation, by investigating the relation between the amount of Technical Debt in new code and the evolution of Technical Debt in the system. To this end, we compare the Technical Debt Density of new code with existing code, and we investigate which of the three major types of code changes (additions, deletions and modifications) is primarily responsible for changes in the evolution of Technical Debt density. Furthermore, we study whether there is a relation between code quality practices and the 'cleanness' of new code. To obtain the required data, we have performed a large-scale case study on twenty-seven open-source software projects by the Apache Software Foundation, analyzing 66,661 classes and 56,890 commits. The results suggest that writing "clean" (or at least "cleaner") new code can be an efficient strategy for reducing Technical Debt Density, and thus preventing software decay over time. The findings also suggest that projects adopting an explicit policy for quality improvement, e.g. through discussions on code quality in board meetings, are associated with a higher frequency of cleaner new code commits. Therefore, we champion the establishment of processes that monitor the density of Technical Debt of new code to control the accumulation of Technical Debt in a software system.

4.1 Introduction

Technical Debt is a metaphor that captures in monetary terms, the cost of additional maintenance effort caused by technical shortcuts taken usually for expediency (Avgeriou et al. 2016). As observed in practice (Digkas et al. 2017), for the majority of software systems, the amount of technical debt increases along evolution, due to growing size and/or reduced

quality; this is aligned with software evolution laws (Lehman et al. 1997). However, the *density* of technical debt, i.e., the normalized amount of Technical Debt per line of code, remains in some cases stable, or is even reduced over time; we have observed this in our previous work on the evolution of the Apache ecosystem (Digkas et al. 2017). This raises the question how such systems manage to maintain or improve their Technical Debt density.

There are two possible explanations for this phenomenon. The first is that these systems follow a process of systematic perfective maintenance, mostly through the application of refactorings (Murphy-Hill et al. 2011). Code refactoring is the most popular strategy for technical debt repayment (Li et al. 2015). However, refactoring activities are rarely applied systematically in practice (Chatzigeorgiou and Manakos 2014, Peters and Zaidman 2012, Arcoverde et al. 2011, Tufano et al. 2017, Tufano et al. 2016). The second explanation is the development of new software artifacts at a level of quality that is above the average, following the clean code paradigm (Martin 2009). Introducing new code whose Technical Debt density is kept below the system average, is not trivial and often implies the adoption of an explicit policy for “clean commits”. The sheer frequency of new commits in large or ultra-large codebases, which in the case of Google’s source code can reach 16,000 on a typical workday (Potvin and Levenberg 2016), renders this strategy even more challenging. Clean code has recently emerged as a promising strategy facilitated by the use of Quality Gates in Continuous Integration systems.

While refactoring has been intensively studied as a technical debt repayment strategy, clean code has not. We argue that if writing clean new code is efficient in managing technical debt in the long term, this has important implications for both practitioners and researchers. The former can scope training activities and software development processes so as to incorporate the best possible practices for writing clean code (e.g., develop quality gates based on the technical debt introduced by new commits). The latter can focus their research efforts on technical debt prevention and repayment within new code chunks.

In this study, we first observe how ‘clean’ is the new code compared to the $TD_{density}$ of existing code for 27 projects from the Apache Software Foundation. Next, we compare the contribution of new, deleted and modified code to the changes in the system’s $TD_{density}$. This allows to understand which activity affects technical debt density the most: writing new code, deleting or modifying code. Finally, we investigate if a decreasing trend in the evolution of $TD_{density}$ is associated with the adoption of relevant practices at project management level. To this end we study whether projects that exhibit a high frequency of cleaner new code: a) provide clear guidelines to committers, so as to guarantee the quality of the newly committed code; and b) consider code quality as an important topic in project management.

We rely on the notion of $TD_{density}$ since absolute measures of technical debt (such as the number of identified violations or estimates of the effort to eliminate these violations) increase monotonically with the addition of code (i.e., absolute measures can decrease only by code deletions). $TD_{density}$ is obtained as the ratio of technical debt effort to remediate the

issues identified in a piece of code, over the corresponding lines of code. In this way, a lower $TD_{density}$ of newly added code might result in a reduction of the system's total $TD_{density}$, even if the technical debt in absolute terms has increased.

In terms of scope, we focus on code technical debt, which is the most studied type of technical debt in the literature (Alves et al. 2016), and the most important type of technical debt in industry (Ampatzoglou et al. 2016). In particular, we consider the TD incurred by code smells in the source code. In terms of granularity, we work at the method level: we monitor the introduction of new methods, and the removal or modification of existing methods. This helps to avoid incorrect classification of code changes during code evolution: changes at the instruction level can become cumbersome to track, as modification, removal and introduction of individual instructions can occur simultaneously. This is further justified in Section 4.3.3. Finally, we emphasize that our scope is open source software, as they provide a long history of commit activity thereby enabling the evolutionary analysis of the study.

The rest of the paper is organized as follows: in Section 4.2 we present related work, i.e., studies that deal with the evolution of software quality and technical debt in particular, empirical studies that provide evidence on the impact and frequency of refactorings as well as recent work on the development of clean new code through the concept of quality gates. In Section 4.3, we present the investigated research questions, the case study design and we discuss how we monitor the contribution of new, deleted and modified code to the system's $TD_{density}$. The results of the study on 27 Apache projects are presented in Section 4.4, while in Section 4.5, we discuss the findings, by providing interpretations and implications for researchers and practitioners. Finally, in Section 4.6 we evaluate the validity of the study, whereas Section 4.7 concludes the paper.

4.2 Related Work

The current study explores the contribution that new code can have on technical debt density, as a complementary approach to applying refactoring. Therefore, this section discusses previous work on: (a) the evolution of code smells and Technical Debt in particular, (b) evidence on the frequency and impact of refactoring, and (c) the concept of Quality Gates that focus on ensuring a desired level of quality in new commits.

4.2.1 Evolution of Code Smells

Lehman's seventh law of software evolution states that *the quality of a system will appear to be declining during its evolution, unless proactive measures are taken* (Lehman 1996). To this end, many studies have explored the evolution of code quality, and if indeed this

law stands in practice. Since this paper focuses on code Technical Debt, we scope this subsection to the evolution of code smells.

One of the first studies that investigate the evolution of code smells was conducted by Olbrich et al. (Olbrich et al. 2009). On their study, they investigate the evolution of two code smells, God Class and Shotgun Surgery, on two projects by the Apache Software Foundation, namely Apache Lucene and Apache Xerces. The results of their study, show that during the software development, there are phases where the number of those code smells can either increase or decrease and those phases are not affected by the size of the systems.

Chatzigeorgiou and Manakos (Chatzigeorgiou and Manakos 2014) have also investigated the evolution of code smells in open-source object-oriented projects. They used historical data of two open-source software projects, namely: JFlex and JFreeChart and studied the evolution of four code smells namely: Long Method, Feature Envy, State Checking, and God Class smells. The results of their study show that as the projects evolve over time the number of code smells tends to increase, which confirms the Lehman's seventh law. Furthermore, they have also found evidence that developers rarely perform targeted refactoring activities to remove smells. In most of the cases, if code smells disappear over time, this is a side effect of regular maintenance (e.g. removal of code). Another interesting finding was that a significant percentage of smells was not the results of software ageing, but smells were present right from the first version of the code in which they reside.

Tufano et al. (Tufano et al. 2017) also studied the evolution of code smells with the goal of understanding when and why code smells are introduced into the projects and observe their life cycle. The study was based on five code smells: Blob Class, Class Data Should be Private, Complex Class, Functional Decomposition, Spaghetti Code. The results indicate that: (a) in the majority of the cases the code smells are introduced into the projects with the creation of the corresponding classes or files, (b) while projects evolve over time, "smelly" code artifacts tend to become more problematic, (c) new code smells are introduced when software engineers implement new features or when they extend the functionality of the existing ones, (d) the developers who introduce new code smells into the projects, are the ones who work under pressure and not necessarily the newcomers, and (e) the majority of the smells are not removed during the project's evolution and few of them are removed as a direct consequence of refactoring operations.

Peters and Zaidman (Peters and Zaidman 2012) studied the lifespans of the following code smells: God Class, Feature Envy, Data Class, Message Chain Class, and Long Parameter List. They developed a tool called SACSEA and used it to analyze the history of eight open-source software projects. Their findings show that while projects evolve, the number of code smells increases. Furthermore, they have also found that although developers are aware of the existence of the code smells they do not perform refactorings. Finally, their findings imply that 'simpler code smells (e.g. Feature Envy Methods) are refactored more often, without any evidence on whether this happens intentionally or not.

Digkas et al. (Digkas et al. 2017) analyzed and tracked the evolution of technical debt of sixty-six open-source Java projects by the Apache Software Foundation, over a period of 5 years. In order to track and detect issues that incur technical debt they relied on SonarQube. The results of their study show that on the one hand, there is a significant increasing trend on the size, complexity, number of Technical Debt Issues, and the total Technical Debt over time, which seems to confirm the software aging phenomenon. But on the other hand, when technical debt is normalized over the non-commented lines of code of the project, an evident decreasing trend over time is present for many of the projects. This could possibly be attributed to: (a) developers that perform refactoring activities and fix some of the open Technical Debt Issues; or (b) developers that introduce better quality code in each commit (compared to the project's existing code base).

Prior research provides evidence that the number of code smells increases over time (Digkas et al. 2017), (Peters and Zaidman 2012) and that smells are often introduced along with the creation of the corresponding classes/files (Chatzigeorgiou and Manakos 2014), (Tufano et al. 2017). However, these studies have not investigated the association between overall trends in system quality with the cleanness of new code or the quality practices followed in a project so as to provide insight into the potential of clean new code as a means of reducing Technical Debt.

4.2.2 Refactoring Frequency and Impact

In this sub-section, we first discuss the frequency at which refactorings are applied, and then we provide evidence on the impact of refactorings on code quality.

Evidence shows that developers rarely apply code refactorings to remove smells. Arcoverde et al. (Arcoverde et al. 2011) studied the lifespan of code smells within software projects and investigated why developers tend to perform very few refactorings. The results of their explanatory survey show that developers are reluctant to perform refactorings in order to avoid API modifications.

Yamashita and Moonen (Yamashita and Moonen 2013) also tried to shed light on why the developers do not perform refactorings on their projects. They conducted an exploratory survey with 85 developers to investigate how familiar they are with the notion of code smells. The results show that one third of the interviewed developers are not aware of code smells or have limited knowledge about them. Furthermore, many of them expressed the lack of good supporting tools that would help them identify smelly pieces of code as candidates for refactoring.

Murphy-Hill et al. (Murphy-Hill et al. 2011) studied broader developers' refactoring habits. Similar to other studies they found that the developers rarely perform refactoring activities and usually, when they do, they combine those refactorings with other code changes. Finally, they observed that even when developers do perform refactoring activities, they do

not systematically record them, e.g. as a message on their commits.

A Google initiative in 2009 asked engineers to participate in a companywide “Fixit” week, focusing on resolving warnings issues by a static analysis tool. Only 16% of the total number of warnings were actually fixed, despite the fact that almost half of the reviewed issues resulted in filing a bug report (Sadowski et al. 2018). It is also noteworthy that Google developers deemed 74% of the issues raised early (i.e. at compile time) as ‘real problems’, compared to 21% of suggested changes for already checked-in code.

A number of studies have empirically investigated the effect of refactoring application on various software qualities. Stroggylos and Spinellis (Stroggylos and Spinellis 2007) examined the logs in the version control systems of four open-source software projects to extract the commits where refactorings had been performed. Next, they measured the effect of refactorings on selected software metrics. The findings reveal that, despite the expectation that refactorings would improve software quality, measurements on the examined systems indicate the opposite. In particular, it was found that refactoring caused a non trivial increase in metrics related to cohesion and coupling.

To investigate how specific quality factors are affected by refactoring, Bois and Mens (Du Bois and Mens 2003) proposed a formalism based on abstract syntax tree representation of source code and projected the impact of refactoring on internal quality metric values defined on this representation. The selected refactorings were Extract Method, Encapsulate Field, and Pull Up Method. Although the study is not focused on obtaining extensive empirical results, the application of the examined refactorings can have a mixed effect on different metrics (such as size, coupling and cohesion ones).

Wilking et al. (Wilking et al. 2007) conducted a controlled experiment to investigate how refactorings affect the maintainability and modifiability of the projects. Their approach consisted in randomly inserting 15 syntactical and 10 non-syntactical errors into code and they measured the time that is needed to fix them. Concerning the effect of the refactorings on the modifiability, they evaluated it by adding new implementation requirements and they measured the time and the Lines of Code that are required in order to implement them. The results of their controlled experiment show that there is no direct effect of improved maintainability or modifiability due to refactoring.

In another study, Alshayeb concluded that refactoring application does not necessarily improve external quality attributes such as adaptability, maintainability, understandability, reusability and testability (Alshayeb 2009). By applying refactoring techniques as defined by Fowler (Fowler 2018) on three software systems and measuring the impact on selected software metrics, an immense variation of the refactoring effect was found. Thus, the author concluded that he was unable to validate that refactoring as a practice improves quality.

A multi-project study on 23 open-source software projects and more than 29000 refactoring operations to study the effect on internal quality attributes was reported by Chavez et al (Chávez et al. 2017). The analysis revealed that 65% of the refactoring operations improve

the internal quality as measured by a wide set of metrics, while 35% of the refactorings keep the quality attributes unaffected.

Although the above set of research studies is not exhaustive, most of the findings agree on the limited adoption of refactorings in practice and a rather mixed effect on software qualities, at least for quality aspects that can be captured by source code metrics. Such evidence calls for the systematic study of other strategies to sustain or improve quality in software systems over time.

4.2.3 Quality Gates

The aforementioned law of declining software quality during software evolution entails that it is not sufficient to write good code in the first place; code has to be *kept clean over time*. As Martin vividly states, this practice adheres to the “Boys Scouts of America” rule to *leave the campground cleaner than you found it* (Martin 2009). The simple and rational strategy of checking-in code that is cleaner than the average of the existing code-base will eventually yield continuous improvement in software quality. In this sub-section we focus on this strategy for reducing $TD_{density}$, i.e., by ensuring that new code commits do not violate a particular set of rules (i.e. do not introduce new Technical Debt Issues) (Martin 2009, Lehman 1996). This strategy is based on the notion of quality gates (Falessi et al. 2017).

Software engineers can use quality gates in order to set constraints, i.e., reject commits that contain any or particular code or design inefficiencies: In case a ‘zero-defect’ policy is adopted, the new code will essentially be TD-free. In practice, quality gates can be more flexible i.e., reject commits that contain smells of a given severity, type or priority level. Quality gates can be easily combined with Continuous Integration (CI) practices setting the maximum level of Technical Debt that is acceptable for new commits to the project’s repository.

Janus et al. in 2012 (Janus et al. 2012) have proposed the 3C Approach. It is an extension to the Agile Practice Continuous Integration and it relies on quality gates for agile quality assurance combining software metrics with Continuous Integration. The proposed automated metric-based Quality Gate checks the source code and ensures that it does not exceed any of the defined thresholds before committing it to the version control system. This way the internal Software Quality is assured. In order to deploy and validate their method, they analyzed an agile project that was developed by a German Automotive Industry company and the results show that a significant improvement of its internal quality can be achieved.

Suryanarayana et al. (Suryanarayana et al. 2015) argued that smells are the result of violating some of the best practices and indicate higher-level design problems. They classified the smells based on the primary object-oriented design principle that they violate, namely: abstraction, modularization, and hierarchyduplicate abstraction, insufficient modularization,

and multipath hierarchy smells. Based on an experiment/study that they conducted they found that one of the reasons that code smells are inserted into the project is the time pressure, thus the developers prefer to perform a quick (and dirty) fix rather than an appropriate solution. Finally, in order to avoid this symptom, they proposed a design quality gate process that checks if the modified/inserted code violates any of the predefined design-level rules.

Schermann et al. (Schermann et al. 2016) acknowledge that Quality gates, as steps that ensure the reliability of code changes, lead to an inherent trade-off between sustaining a fast pace and risking a lower release quality. To address this issue they proposed a model where software releases are evaluated based on the Confidence (reliability) and Velocity (publishing speed). Their Confidence-Velocity Categorization Model consists of the following four categories: Cautious (low Velocity and high Confidence), Balanced (high Velocity and high Confidence), Problematic (low Velocity and low Confidence), and Madness (high Velocity and low Confidence).

Nevertheless, according to the empirical investigation by Vassallo et al. (Vassallo et al. 2018) Continuous Code Quality (CCQ) is not applied in practice. The authors attribute the low use of CCQ to the fact that code quality is not always the top priority for development teams but also the unawareness of how to properly set up quality gates.

4.3 Case Study Design

Case study is an empirical method that is used for studying phenomena (e.g., projects or activities) in a real-life context (Wohlin et al. 2012). The case study of this paper has been designed and is presented according to the guidelines of Runeson et al. (Runeson et al. 2012).

4.3.1 Goal and Research Question

The goal of this study is to compare addition, deletion and modification of code regarding their impact on $TD_{density}$. Moreover, to provide further insight to the relevant strategies, we study whether code quality practices are associated with the cleanness of new code. Therefore, we formulate two relevant research questions.

RQ₁: Among the three major types of code changes (insertion, deletion and modification) which is primarily responsible for changes in code $TD_{density}$?

RQ₁ aims at investigating whether changes in technical debt density from one code revision to the next are primarily associated with addition of new code, deletion or modifications of existing code. Each type of change can incur a negative or positive effect on the system's technical debt density depending on the quality of the code that is added, modified or removed. Code modifications can sometimes be related to the application of refactorings, but in the general case we assume that code changes are the result of maintenance and not necessarily targeting the removal of inefficiencies.

RQ₂: Is the frequency of commits in which new code is cleaner compared to exiting code, associated with the existence of practices targeting code quality?

RQ₂ aims at investigating whether a high percentage of cleaner new code commits is related to the use of practices targeting code quality. In terms of relevant practices we study two project management aspects: (a) the existence of commit guidelines (i.e. what the developer should have in mind before committing his/her code) which are directly or indirectly related to the avoidance of TD rule violations; and (b) the extent to which quality related issues (e.g., code improvement, code quality, refactorings, etc.) are discussed in project board meetings. Assessing the strength of the association can lead to interesting actionable outcomes, which can guide project managers on how to control the quality of their projects.

4.3.2 Cases and Units of Analysis

This study is characterized as multiple, embedded case study (Runeson et al. 2012), in which the cases are open-source software (OSS) projects and the units of analysis are the revisions across the project history; we analyse changes to the system's $TD_{density}$ in these revisions. The reason for selecting to perform this study on OSS systems is the vast amount of data that is available in OSS repositories, in terms of revisions and classes, as well as quality-related practices. The long history that is available for each OSS project enables researchers to observe overall trends in the evolution of their quality. To retrieve data from only high-quality projects that evolve over a period of time, we have selected to investigate the projects presented in Table 4.1. We have decided to focus on Apache projects (similarly to the studies by Tan et al. (Tan et al. 2018) and Tufano et al. (Tufano et al. 2017)) since the Apache Software Foundation, as an OSS development organization, has a reputation for high quality projects, for putting emphasis on process and quality improvement as well as for long-lasting projects.

The project selection process was based on the following criteria:

- a. The project should be active (based on the date of its last commit) and therefore still maintained. This criterion aims at ensuring that the analyzed projects are still undergoing development and thus the studied practices are up-to-date. A similar prerequisite has been set by Rausch et al. (Rausch et al. 2017) who studied the build failures in Continuous Integration (CI) workflows of open-source software.
- b. The software should be written in Java and use Maven as a build tool. This ensures that the project can be built and allows the retrieval of the project's language version from the corresponding pom.xml file.
- c. The software should contain more than 500 classes to ensure the inclusion of systems with a substantial size, functionality and complexity. A minimum number of system

Figure 4.1: Seeking the longest path between commit nodes

classes has also been set as a project selection criterion in the studies by Tan et al. (Tan et al. 2018) and Olbrich et al. (Olbrich et al. 2009).

- d. The software should have more than 1000 commits and should be under development for at least 3 years. We have included this criterion for similar reasons to the previous criterion and to be able to observe trends in the evolution of the projects quality. Moreover, this number of revisions provides an adequate set of repeated measures as input to the statistical analysis. A minimum number of commits has also been used as a criterion in other studies on software evolution (Rausch et al. 2017, Tan et al. 2018, Olbrich et al. 2009, Peters and Zaidman 2012, Herraiz et al. 2006).

The selection of Apache projects enabled us to perform the analysis on RQ₂ which is based on the availability of minutes for Apache Board Meetings. However, we should note that there are also other Apache projects fulfilling the above mentioned criteria beyond those included in our dataset. Due to the complicated data analysis we have excluded projects that are extremely large, either in the number of classes or the number of commits.

4.3.3 Tracking the Types of Changes

Considering that software systems evolve through a number of revisions and that in each revision several types of changes may occur simultaneously, we look at the three major types of method changes: the development of new methods, the deletion or the modification of existing ones. These primary types of evolutionary changes have been considered in other studies as well (Beyer et al. 2017, Jaafar et al. 2016, Xing and Stroulia 2005, Dagenais and Robillard 2011). As already mentioned, monitoring changes at the instruction level would be more complex and less accurate considering that several types of changes can simultaneously occur in some statements (e.g., modification and introduction of new code). Furthermore, tracking changes at the instruction level is challenging, as one would have to map each instruction (in a particular revision) to the corresponding instruction in the previous revision. This process is complicated by the insertion of new statements, comments, blank lines, etc. Therefore, to be certain about the classification of changes, we monitor changes at the method level.

Similarly, instead of assessing the entire technical debt of the analyzed systems, i.e.,

Table 4.1: Selected Projects

Project	Classes	NCLOC	Analyzed Revisions
Accumulo	5840	428543	2863
Atlas	932	87637	1454
Beam	3757	176663	2780
Calcite	2606	186633	1448
Cayenne	2615	164170	2116
CXF	4111	353085	5079
DeltaSpike	951	46182	513
Drill	4655	316552	1316
Dubbo	943	61865	728
Flink	5632	341149	5329
Flume	790	51897	789
Giraph	1414	72972	668
Jackrabbit	2883	273574	4260
jclouds	5687	227459	4323
Knox	1083	51429	1033
Kylin	1658	128531	3205
Metron	1433	72579	548
MyFaces	1843	174158	1211
NiFi	4256	371031	1490
oozie	1082	97597	587
OpenWebBeans	561	44299	1583
PDFBox	1279	136916	3758
Pulsar	1837	147182	1503
SIS	1948	181588	828
Storm	3958	243574	738
TinkerPop	1698	95652	5178
Zeppelin	1209	89193	1562

considering violations on every individual line of code, we have opted to consider only TD that can be mapped to class methods. In other words, we consider only SonarQube rule violations which reside in class methods. The reason is that Technical Debt Issues which occur at the class- or file-level (e.g., “The default unnamed package should not be used”) are not associated with particular lines of code; as a result it would not be possible to assess what kind of code change caused their introduction or removal.

At each revision a method can be added, deleted, modified or remain unchanged. According to the stated research questions, the goal of this study is two-fold: (a) since multiple types of changes might occur simultaneously, to identify the type of change that has the largest impact on $TD_{density}$ change, i.e. whether the $TD_{density}$ change from one revision to the next is mostly due to the modification, the deletion or the addition of new methods, and (b) to investigate whether the frequency of ‘clean’ new code commits is related to overall project policies.

4.3.4 Data Collection

To analyze the projects and measure Technical Debt throughout their evolution, we have used SonarQube. SonarQube relies on a set of rules which are checked by static source code analysis; every time a piece of code breaks one of those coding or design rules, a technical debt issue is raised. Thus, SonarQube estimates the effort (in minutes) required to eliminate the identified Technical Debt issues¹. This effort is obtained by assigning a time estimate for fixing each type of problem and by multiplying all issues of the same type with that estimate. It should be noted that the Apache Foundation ecosystem, has a dedicated SonarQube instance for quality control in its projects. Currently, 336 Apache projects are continuously monitored through SonarCloud, and 90.1% pass the quality criteria set by the development teams². SonarQube reports various types of problems, namely code smells, bugs (issues representing something wrong in the code), vulnerabilities, code duplications and lack of test coverage. We note that in this study, we only consider code smells, since the other two types of problems (i.e., bugs and security vulnerabilities) do not fit the definition of TD (Avgeriou et al. 2016), in the sense that they do not concern maintainability or evolvability.

For RQ_1 we measure the contribution to the $TD_{density}$ change at each revision that is due to (a) new methods, (b) removed methods, and (c) modified methods. For RQ_2 we consider the revisions in which the $TD_{density}$ of new code is lower than that of existing code.

Data Collection for Answering RQ_1

We have devised a process for analyzing git repositories which can be outlined in the following phases and individual steps:

Phase 1: Retrieval of commits

1. First, the Git history for the project under study is retrieved from its master (default) branch since it reflects the production-ready state of the project.
2. All commits are sorted to form a time series of revisions that have been performed on the source code. This process is non-trivial since even on the master (default) branch

¹In this study we have considered Technical Debt issues reported as code smells by SonarQube

²<https://sonarcloud.io/organizations/apache/projects>

a commit can have multiple parents. To treat the data as a single time series, a single parent should be chosen for each commit without forming any branches. For the cases of commits with more than one parent, we have employed an algorithm aiming at identifying the longest path between the commit node under examination and the start node (i.e. the only node with no parent). As an example, Fig. 4.1 shows that commits 13 and 15 have more than one parents. We select for analysis the path consisting of the black nodes since it forms the longest possible path. Had we selected the series of commits as formed chronologically we would have run into inconsistencies among revisions: for example, if we analyze CM1, CM2, CM3, CM4, CM5 and so forth, any change on Technical Debt at commit CM4 would not be valid for the chronologically subsequent commit CM5, and changes to TD across revisions would yield irrational results. At the same time, the longest path yields the largest number of commits to be analyzed and thus results in a higher granularity for the analysis.

3. To reduce the computation time, a filtering step is applied: we ignore transitions between successive commits that do not involve any changes to Java files. We do this because the analysis of multiple revisions of large projects in SonarQube is extremely computationally-intensive resulting in several hours or even days for analyzing the entire history of the selected open-source projects.
4. From all commits submitted for analysis to SonarQube we retain only the successfully analyzed commits. The reason is that several commits may fail to analyze for various reasons, such as an incorrect pom.xml file that prohibits the build of the project.

Phase 2: Mapping of Technical Debt Issues to methods

To map the identified Technical Debt Issues to the class methods of each revision we perform the following steps:

1. First, for each revision, we retrieve all Technical Debt Issues by performing the corresponding query to the SonarQube database.
2. Next, we map the identified Technical Debt Issues to the methods of the corresponding revision. This is performed by matching the line in which each Technical Debt Issue is reported by SonarQube (in case the Technical Debt Issue concerns multiple lines, SonarQube reports the first one) with the method containing that line.

Phase 3: Tracking method changes

In order to associate variations in the overall $TD_{density}$ of a system with code changes at the method level, we track the type of change (introduction, deletion, modification) occurring to each method as follows:

1. For the new and deleted files of each revision (obtained from git history) we obtain their representation in the form of an Abstract Syntax Tree (AST)³. For each new/deleted file, we extract all its methods from the AST representation and then tag all these methods as new/deleted, respectively.
2. For the modified files of each revision we track new/deleted/modified/unchanged methods in each transition with the help of the Gumtree Spoon AST Diff tool (Fallieri et al. 2014).

Phase 4: Calculating the contribution of new/deleted/modified methods to the change in the system's $TD_{density}$

Finally, we need to calculate, for each revision in the system's history, the contribution of new/deleted/modified methods to the change of the system's $TD_{density}$. Let us consider a transition from revision $t-1$ to revision t . To segregate the contribution of each type of change and at the same time ensure that the sum of all contributions is equal to the total change in the system's $TD_{density}$, we subtract the $TD_{density}$ of the previous revision from the $TD_{density}$ that is derived by the addition, removal or modification of code. The calculation is outlined in the following formulas:

Contribution of new methods

$$\Delta TD_{density}(new) = \frac{TD_{t-1} + TD_{new(t)}}{LOC_{t-1} + LOC_{new(t)}} - TD_{density}(t-1) \quad (4.1)$$

Contribution of deleted methods

$$\Delta TD_{density}(deleted) = \frac{TD_{t-1} - TD_{deleted(t)}}{LOC_{t-1} - LOC_{deleted(t)}} - TD_{density}(t-1) \quad (4.2)$$

Contribution of modified methods⁴

$$\Delta TD_{density}(modified) = \frac{TD_{t-1} \pm \Delta TD_{modified(t)}}{LOC_{t-1} \pm \Delta LOC_{modified(t)}} - TD_{density}(t-1) \quad (4.3)$$

As a result, the change in the system's $TD_{density}$ is equal to the sum of the individual contributions:

³The AST is obtained through the Eclipse Java Development Tools (JDT)

⁴In eqs. (2), (3) the denominator can not obtain the value zero under real circumstances, as this would imply that all lines of code are deleted or modified in a certain revision

$$\begin{aligned}
\Delta TD_{density}(system) = & \Delta TD_{density}(new) \\
& + \Delta TD_{density}(deleted) \\
& + \Delta TD_{density}(modified)
\end{aligned} \tag{4.4}$$

It should be noted that the data collection process has led to an enormous data set of approximately 1.4TB. A replication package with all data required to study the two RQs is available online⁵.

Data Collection for Answering RQ₂

To explore the association of code quality practices and the quality of new code (RQ₂), we use the results of the descriptive statistics (the percentage of commits in which the new code is cleaner compared to existing code [CLEAN_CODE_FREQ]), and two other variables. The first variable [COMMIT_GUIDELINES] is binary, and is set to true if: (a) the website of the project has clear and public guidelines for committers (usually termed “How to ...”); and (b) if at least one of the guidelines is not a purely aesthetic/formatting guideline (e.g., indent your code using tabs) and is directly or indirectly related to the rules being checked by SonarQube. A detailed reporting of how each project has been evaluated with respect to Commit Guidelines is presented in the online replication package. The second variable [PROJECT_BOARD_MEETINGS] is related to the emphasis of the project board on quality issues. In particular, for each Apache Software Foundation project, there is a regular meeting (usually every 3 months), in which the managers or key contributors of the project discuss the open issues and strategies for further improvement. To assign a value to [Project Board Meetings] variable, we have parsed the minutes of these meetings, aiming to identify discussions related to:

- *quality control (QC)*, for which we searched for the keywords: “software quality”, “code quality”, “code improvement”, “code review”, “guideline”, or “sonar”
- *refactorings (REF)*, for which we searched for the occurrence of “refactoring” and “clean up”

and recorded the number of meetings in which each term was identified (variables [QC] and [REF]). Next, we calculated and rounded the MEDIAN value for the two variables ([QC] and [REF]). Every value that was higher than the rounded median was characterized as HIGH, whereas the rest as LOW. Projects characterized as HIGH in both perspectives, have been marked as HIGH in the [Project Board Meetings] variable, whereas all the rest as LOW. In

⁵Replication package is available at https://drive.google.com/drive/folders/1mxher2vkE68GzKAkz1Y7rjVB0_hTihzt

other words, we classify projects in two categories based on the frequency by which project board meetings deal with code quality and refactoring strategies.

4.3.5 Data Analysis

To answer the research questions using the collected data we carried out both descriptive and inferential statistical analysis as follows:

Data Analysis for Answering RQ₁

The investigation of the contribution of each type of code change to the variation of the system's $TD_{density}$ across revisions is quite complicated, as the effect of the three types of changes (additions, deletions and modifications of methods) has to be taken into account. In particular, we are interested in observing whether positive (negative) changes in the system $TD_{density}$ co-exist with positive (negative) contributions stemming from new/deleted/modified methods.

Independent Variables

Contribution to $TD_{density}$ of new, deleted and modified code. These are categorical variables.

Categories: Leading to a decrease, increase or no change (stable) in $TD_{density}$.

Dependent Variable

Direction of $TD_{density}$ change during a transition from one revision to the next (the cases when the $TD_{density}$ remained stable are rare and are omitted for clarity). This is a categorical variable.

Categories: Increase/Decrease.

Analysis

We first obtained contingency tables to describe the relationship between the two categorical variables. It should be noted that to investigate the effect of each type of change, we retained only the transitions when two types of changes occurred simultaneously, that is when two types of changes *compete* for the effect on the overall change in $TD_{density}$. In case only one type of change has occurred during a transition, then it is obvious that the change in $TD_{density}$ will be the result of this single change, and thus including such transitions in the data set would lead to misleading results. The results are displayed in the form of heat maps. For each project (row) three individual heat maps are shown, one for the contribution of new, deleted and modified code, respectively.

To further investigate this relationship, we performed a chi-squared test between the two categorical variables, to determine whether there is a significant relationship between them.

Null Hypothesis H_0 : assumes that there is no relationship between the direction of change in the system's $TD_{density}$ (decrease or increase) and the corresponding direction of change caused by new, deleted or modified methods.

Alternative Hypothesis H_1 : Assumes that there is an association between the two variables.

Finally, to shed light into the effect of new methods vs. the effect of modified methods on code improvement, we illustrate graphically (in a bar chart) the percentage of transitions in which a reduction in the system's $TD_{density}$ co-occurred with positive contributions (i.e. leading to a decrease of $TD_{density}$) by new and modified methods, respectively.

Data Analysis for Answering RQ₂

To answer RQ₂, we explored whether: (a) projects that provide commit guidelines or (b) projects in which Project Board meetings often refer to code quality, are having a statistically significant higher average number of commits of cleaner code, compared to projects that do not provide guidelines and do not discuss code quality often.

Independent Variables

Binary variable [COMMITGUIDELINES] representing whether a project has commit guidelines related to code quality. Values: YES, NO.

Binary variable [PROJECTBOARDMEETINGS] representing the frequency by which quality issues are discussed in project board meetings. Values: LOW, HIGH.

Dependent Variable

The percentage of commits in which the new code is cleaner compared to existing code [CLEAN_CODE_FREQ].

Analysis

We explored the discriminative power of the [COMMIT_GUIDELINES] and the [PROJECT_BOARD_MEETINGS] in terms of the [CLEAN_CODE_FREQ] variable. To this end we have used boxplots to illustrate any differences in the percentage of cleaner code commits between projects that do not provide commit guidelines vs. those that provide them, and between projects that often refer to code quality issues vs. those that do it less often. Moreover, we have performed independent samples t-test to test any statistically significant differences. being

Null Hypothesis H_0 : assumes that there is no difference in the percentage of cleaner code commits, regardless of any adopted code quality practices (means are equal).
 Alternative Hypothesis H_1 : Assumes that the percentage of cleaner code commits differs depending on the adopted code quality practices (means are not equal).

4.4 Results

In this section we present the results of our study organized by research question, and highlight the major findings. However, prior to answering the research questions, we present a visualisation and descriptive statistics on the $TD_{density}$ of individual commits for the selected projects. This can provide the context upon which we can interpret the results of the research questions.

4.4.1 Descriptive Statistics

To obtain a first insight into the quality of new code as opposed to the quality of the system in which the new methods are added, we plot the evolution of the system's $TD_{density}$ along with the $TD_{density}$ of individual commits where new methods are added.

Figure 4.2 illustrates for one project (Commons IO)⁶ the evolution of the system's

⁶Project Commons IO is used as a motivating example and has not been included in our dataset since it is rather

Figure 4.2: Contribution of new code on system's $TD_{density}$ (Motivating Example)

Table 4.2: Percentage of revisions in which new methods have lower $TD_{density}$, for all examined projects

Project	%	Project	%	Project	%
Accumulo	65	Flink	77	NiFi	81
Atlas	73	Flume	73	oozie	79
Beam	71	Giraph	74	OpenWebBeans	77
Calcite	86	Jackrabbit	81	PDFBox	77
Cayenne	82	jclouds	84	Pulsar	72
CXF	77	Knox	69	SIS	68
DeltaSpike	72	Kylin	71	Storm	61
Drill	84	Metron	84	TinkerPop	75
Dubbo	81	MyFaces	71	Zeppelin	72

$TD_{density}$ (black dots) and the corresponding trend-line, depicting a gradually increasing quality (black line declines over time). On the same plot, blue dots correspond to the revisions, in which the $TD_{density}$ of new methods was lower than that of the system in that revision, while red dots indicate the cases where the $TD_{density}$ of new methods was higher⁷. As it can be observed, for the vast majority of revisions (77%), the $TD_{density}$ of new methods is lower than the $TD_{density}$ of the host system and in many cases the new code is entirely TD-free (see blue dots on the x-axis).

As it is not possible to show similar plots for all projects, Table 4.2 shows the percentage of revisions for which the $TD_{density}$ of new code was lower than the $TD_{density}$ of the system in the corresponding revision. The findings confirm the first impression that new code in the examined systems is generally of a higher quality than the existing baseline: in the majority of the commits, new methods have lower $TD_{density}$ than the host system. Considering that many of these systems have a quality that increases over time, it would be reasonable to argue that the cleanness of new code has contributed to the declining trend of the system $TD_{density}$. However, to claim that new code is a prominent factor that leads to the reduction of TD requires further analysis.

Moving on to more detailed descriptive statistics, the boxplots in Fig. 4.3 and Fig. 4.4 illustrate the distribution of the difference between the $TD_{density}$ of new methods and the $TD_{density}$ density of the host code. To allow for a fair comparison, we differentiate between the case when one or more new methods are introduced in an existing class and the case

small and has a limited number of revisions

⁷For clarity, an upper bound on the displayed $TD_{density}$ values is imposed, that is, data points with an extremely high $TD_{density}$ are not accurately depicted but are simply placed on the upper bound (top of the figure).

Figure 4.3: Distribution of the difference between the $TD_{density}$ of new methods (introduced in new classes) and the $TD_{density}$ of existing system, for all projects

when a set of new methods are introduced in the form of a new class. In the former case the $TD_{density}$ of the new methods should be contrasted against that of the class in which they are added, since the class resembles the *neighborhood* of the new code, in terms of functionality and complexity. For the case of completely new classes, the comparison should be made against the entire system in which the new class is added, as the system is the *neighborhood* of the introduced class. The boxplot of Fig. 4.3 shows the distribution of the difference in $TD_{density}$ between new methods added in *new* classes at revision i and the quality of the entire system in the previous revision ($i-1$). The boxplot of Fig. 4.4 shows the distribution of the difference in $TD_{density}$ between new methods added in *existing* classes at revision i and the quality of the class in which they are added, in the previous ($i-1$) revision.

As it can be observed from the boxplots, the median difference (μ) between the $TD_{density}$ of new methods and that of the host code, is negative for all but one projects. For most of the projects, and especially for new methods introduced in existing classes, even the upper quartile is below zero. Thus, it becomes evident that in the transitions in which new code was added (in the form of entirely new methods either in new classes or in existing classes) the $TD_{density}$ of the new code is significantly lower than that of the host code while in many cases it is very close to zero.

Figure 4.4: Distribution of the difference, between the $TD_{density}$ of new methods (introduced in existing classes) and the $TD_{density}$ of existing system, for all projects

4.4.2 Relation among the contribution of new / deleted / modified methods and change in system's $TD_{density}$ (RQ₁)

We remind that for studying this RQ we created two categorical variables: the first refers to the direction of change (decreasing/increasing⁸) of the system's $TD_{density}$ in each revision. The other refers to the contribution (decreasing, increasing, stable) of new/deleted/modified methods in each revision. The contribution itself is calculated according to equations (1)-(3). We have turned this contribution to a categorical variable depending on whether the contribution is positive, zero, or negative.

The results from the cross-tabulation of frequencies between these two variables are displayed in Table 4.3 for all the analyzed projects. For each project (composite row) three individual heatmaps are presented, one for the contribution of new, deleted and modified code respectively. Each heatmap is comprised of six cells: the two rows correspond to the increase or decrease in the system's $TD_{density}$, whereas the three columns to the effect (decrease, increase and stable) of the corresponding change type (new, deleted or modified). The intensity of the color (within each 6-cell heatmap) indicates the frequency of occurrence for each combination between the two categorical variables, that is, the direction of change in the system's $TD_{density}$ (decrease or increase), and the direction of change (decrease, in-

⁸The cases where the system's $TD_{density}$ remained stable are rare and are omitted for the sake of simplicity

Figure 4.5: Percentage of revisions in which a decrease in the system $TD_{density}$ co-occurred with a positive contribution in quality by new and modified methods

crease and stable) for each type of contribution. It should be noted that while absolute numbers (number of transitions in which each combination has been observed) are shown on the heatmaps, the intensity of the colors reflects the corresponding percentage of cases (highest percentage corresponds to the most intense red).

Let us consider as an example, the first project in Table 4.3 (project `Accumulo`). We focus on the contribution of new code (composite column `New`) and study separately the two rows, corresponding to transitions where the system’s $TD_{density}$ has decreased and increased, respectively:

- *System $TD_{density}$ has decreased (top row):* The warm (red) color in the upper left cell (labeled with 195) implies that among the cases where new code was added and the system $TD_{density}$ has decreased, the highest frequency (195 out of the total 242 transitions) was observed for new methods that contributed to a decrease in the $TD_{density}$.
- *System $TD_{density}$ has increased (bottom row):* The red color in the lower center cell (labeled with 113) implies that among the cases where new code was added and the system $TD_{density}$ has increased, the highest frequency (113 out of the total 210 transitions) was observed for new methods that contributed to an increase in the $TD_{density}$.

In other words, in this project, the change in the system $TD_{density}$ co-occurred in most of the cases with a contribution of the same direction by new code.

Following this kind of interpreting Table 4.3, for the contribution of new methods in all projects, it can be observed that when their contribution leads to a reduction of $TD_{density}$, in most of the revisions the same direction of change is observed in the system’s $TD_{density}$ (warm red color in the top-left cell in each six-cell heatmap). In other words, in most of the cases, **when new code is *cleaner*, the system’s $TD_{density}$ decreases**. However, an impact on the system’s $TD_{density}$ cannot be claimed when new code contributes to an increase of the $TD_{density}$. It should be noted that this pattern is consistent among all projects. For deleted methods the most striking observation (most red cells) concerns the cases when the deleted methods contribute to an increase of the $TD_{density}$ (for example, when high quality code is

removed from the system). In those cases it seems that **the deletion of high quality code most frequently co-exists with an increase in the system's $TD_{density}$** .

An interesting and repeating pattern is present for modified methods. Intense red colors are observed in alternating rows: this implies that the direction of change in the system's $TD_{density}$ coincides with the contribution of modified code. In other words, **if a method is modified and the $TD_{density}$ of the method decreases, then, for the majority of the cases a decrease in the system's $TD_{density}$ is observed; similarly for an increase in the $TD_{density}$** .

Finally, as expected, the lack of any contribution of new/deleted/modified methods (column $-$) does not appear to have any association with the overall change in the system's $TD_{density}$ as depicted by the mostly blue cells. It should be emphasized, that these observations, should by no means be interpreted as indications of causality. Investigating whether each type of change (new/deleted/modified code) is responsible for the change in the overall $TD_{density}$ would require a different experimental set up and is beyond the scope of this study.

The chi-square test for independence has been used to discover if there is a relationship between the direction of change in the project's $TD_{density}$ and the contribution of new/deleted/modified code. Table 4.4 shows for each project the Pearson chi-square value (top row) and whether the results are statistically significant or not depending on the p-value. The bottom row for each project shows the Phi value that tests the strength of the association (Pace 2012).

As it can be observed, in almost all cases the results are statistically significant ($p < 0.01$) implying that the null hypothesis can be rejected. In other words we can argue that there is a relationship between the direction of change in the systems $TD_{density}$ and the contribution of new/deleted/modified code. In particular, **the strength of the association appears to be higher for the contribution of modified methods (for most projects), followed by the contribution of new code. The contribution of deleted methods seems to have a significantly lower association to the change in the system's $TD_{density}$** .

These results concern both directions of change in the system's $TD_{density}$ and reveal that code modification can contribute positively and negatively to changes in the system quality. If we focus only on the cases where the $TD_{density}$ decreased from one revision to the next, the potential of cleaner new methods becomes more evident: The barchart of Figure 4.5 displays the percentage of transitions where a decrease of the system's $TD_{density}$ was observed and new/modified methods also contributed to a decrease of $TD_{density}$ (assuming that at least two types of changes were competing in the same transition). The cases where a positive contribution by new methods co-occurred with an improvement in the overall quality, are slightly more frequent.

4.4.3 Relation between Code Quality Practices and New Code Clean-ness (RQ₂)

In Table 4.5, we present the data extraction results for the studied projects, with respect to the employment of code quality practices at project management level.

With respect to the existence of commit guidelines, we can observe that 16 projects provide guidelines related to TD rule violations, and 11 projects do not. By comparing the median values of the percentage of cleaner code commits (see Figure 4.6) in the two groups, we can observe that projects that provide commit guidelines are having more commits (an increase of the median by 3.88% has been observed) in which the new code is cleaner compared to existing code. However, this difference is not statistically significant, based on the independent samples t-test (sig = 0.09).

Figure 4.6: Distribution of the percentage of cleaner code commits for the two groups of projects, based on the existence of commit guidelines

On the other hand, the 9 projects in which the project management team more regularly discusses code quality in the board meetings, are having a statistically significant

higher percentage of commits in which the code is cleaner ($\text{mean}_{diff} = 5\%$, and $\text{sig} = 0.05$). This difference is visualized in the boxplots of Figure 4.7, in which we can observe no overlap in the boxes (Q3-Q1) of the two groups.

Figure 4.7: Distribution of the percentage of cleaner code commits for the two groups of projects, based on the frequency of references to code quality in board meetings

4.5 Discussion

In this section we first discuss in detail the findings presented in Section 4, attempting our own interpretation. Then, we list potential implications for researchers and practitioners.

4.5.1 Interpretations of the Results

Quality of new code vs. system quality (Descriptive Statistics)

The descriptive statistics revealed that new code (in the form of new methods), exhibits quality (in terms of $TD_{density}$), that is higher compared to the quality of the systems in which the new methods are introduced. This finding was consistent in all examined projects. For many of the projects in the Apache Software Foundation ecosystem this could be linked to the observed declining trend of the project's $TD_{density}$, although this study cannot establish causality between cleaner new code and improvement of overall quality. It still remains to be studied how developers ensure that new code is cleaner in terms of technical debt. It might be a deliberate choice (e.g., in case the development team applies a Quality Gate to ensure zero or low number of violations for each new commit) or a general trend resulting from improvements in the employed processes and tools or simply the result of higher developer experience.

It is also noteworthy that in a very large percentage of the analyzed commits, new methods had zero or very low $TD_{density}$. In particular, the findings from the selected projects, suggest that the overall system quality can improve (i.e. $TD_{density}$ can decrease) over time if the $TD_{density}$ of new code is systematically kept below the system's average. We caution, that this does not imply a quick fix to a system's quality (in terms of TD); if the existing code base is largely of low quality, the improvement that can be achieved through new commits is limited in the short term. Nevertheless, the improvement of $TD_{density}$ in new code has merit in the sense that most systems evolve for years; thus, in the long run, systematically writing cleaner new code can yield substantial code improvement. This observation, emphasizes the importance of writing clean new code over the practice of refactoring. We also note that several empirical studies have shown that systematic, bad smells refactoring is uncommon in most open-source software projects, rendering the practice of writing clean new code even more valuable. This result can be of particular worth, in the sense that writing clean code can be a best practice for limiting the *software ageing* phenomenon (Parnas 1994) or the 7th of Lehman's laws of evolution which states that the quality of software deteriorates over time (Lehman 1996).

Relation among the contribution of new / deleted / modified methods and change in system's $TD_{density}$ (RQ₁)

The in-depth study on the association between the contribution of new / deleted / modified methods and the observed changes in the system's $TD_{density}$ revealed some interesting patterns: (a) among all transitions where the system's $TD_{density}$ has decreased and two or more types of changes were competing, the most frequent case was the addition of new methods that were better in terms of $TD_{density}$, implying that the introduction of clean new code

coincided with an improvement in quality; (b) the contribution of method removal is rather mixed, which is reasonable, as the effect of method deletion on the system's $TD_{density}$ depends on the quality of the removed code (the removed methods can be either high or low quality code); and (c) the contribution (positive or negative) of method modification coincides with the direction of change in the system's $TD_{density}$. For the latter case it should be emphasized that modifications refer to any type of changes to a method including adaptive and corrective maintenance.

The chi-square test of independence showed vividly that there is a statistically significant relationship between the direction of change in the system's $TD_{density}$ and the contribution of new, deleted and modified code. Considering that code deletion is not usually performed on the basis of quality improvement but rather dictated by functionality-related reasons, the improvement of code quality is subsequently left up to the addition and modification of code. Method modification has a clear association with the overall technical debt: lowering the $TD_{density}$ of a method during maintenance will reduce the system's $TD_{density}$ and vice-versa. On the other hand, cleaner new methods (which according to the descriptive statistics are very common) have a strong association to a decreasing system $TD_{density}$. These results, further emphasize the importance of writing high-quality new code and monitoring the introduced number of Technical Debt Issues in new code. To provide insight into how TD can be eliminated or avoided during software evolution, we provide two real examples: the first refers to code modification and the second to the introduction of clean new code.

The first example (**code modification**) refers to a pull request in project Dubbo (#3474) where the purpose of the change was to properly close resources after use, thereby eliminating an existing TD issue. The intention is also reflected on the title of the pull request: 'Fix Not Properly Closed Resources'. The affected code, prior to the change, was:

```
try {
    UnsafeByteArrayInputStream is = new
    UnsafeByteArrayInputStream((byte[]) args[i]);
    . . .
} catch (Exception e) {
    . . .
}
```

and thus was missing a proper close call within a finally block, causing a Blocker issue according to SonarQube. The change in the pull request targeted exactly that problem and fixed it by using the try-with-resources statement, that declares the resources to be closed after the program is finished (which is equivalent to using a finally block prior to Java SE 7). The corresponding code was modified to (note that the resource is declared within parentheses after the try keyword):

```
try (UnsafeByteArrayInputStream is = new
    UnsafeByteArrayInputStream((byte[]) args[i])){
```

```

    . . .
}
catch (Exception e) {
    . . .
}

```

thereby, eliminating the abovementioned SonarQube issue.

The second example (**of clean new code introduction**) refers to a pull request in the same project `Dubbo` when new methods are introduced in the new `NettyClientHandler` class. Up to that point, the code suffered from multiple TD issues related to improper handling of Exceptions, violating the major ‘Throwable and Error should not be caught’ rule 349 times. Noncompliant code examples are of the following form:

```

try { /* ... */ } catch (Throwable t) { /* ... */ }
try { /* ... */ } catch (Error e) { /* ... */ }

```

The rationale for this rule is that `Throwable` is the superclass of all errors and exceptions in Java, while `Error` is the superclass of all errors, which are not meant to be caught by applications. Catching either `Throwable` or `Error` will also catch `OutOfMemoryError` and `InternalError`, from which an application should not attempt to recover⁹. Class `NettyClientHandler` in pull request #630 is TD free and while it deals with exception handling in all of its methods, none of the methods violates the aforementioned rule.

Code Quality Practices and Cleanness of New Code (RQ₂)

We observed that projects in which Code Quality is often being discussed among the management team, exhibit a statistically significant higher percentage of commits with code that is cleaner than the existing codebase. These discussions are sometimes very explicit about the use of tools to measure code quality or the emphasis on cleaning up the code. As an example, in an Apache `PDFBox` board meeting of 2015, under ‘Software Quality’ it is mentioned that ‘*There is an ongoing effort to improve PDFBox based on the analysis of different tools such as SonarQube, FindBugs and others*’. In a March 2019 meeting of project `Flink` under ‘Status’ it is noted that ‘*The release contains some new user-facing features plus a lot of internal cleanup and refactoring, fixing some long term issues ...*’.

Apache Project Management Committees (PMCs) are required to report on their project’s health and status quarterly to the Board of Directors. We have observed in our study that projects with a certain level of size and complexity, code quality in general and maintainability in particular becomes a major concern. Guiding the hundreds of volunteers in open-source projects on how to commit high quality code can be facilitated by the use of tools/practices

⁹<https://rules.sonarsource.com/>

which essentially dictate a minimum threshold of quality that has to be reached before submitting code. We argue that such a ‘clean new code’ policy is also applicable to industrial projects as a means of sustaining, and even improving TD.

4.5.2 Implications for Practitioners

Regarding software developers, evidence from the presented case study suggests that new code can have a substantial impact on the quality of an evolving system. The fact that the contribution of new code has a strong association to the changes in the system’s $TD_{density}$ implies that writing clean new code can ensure, to a large extent, the gradual improvement of the overall system quality. Of course, code modifications that result in lower $TD_{density}$, either in the form of refactorings or as carefully applied maintenance, has also potential for improving code quality.

In terms of software development strategies, we believe that using Quality Gates to enforce a predefined quality policy can be a simple, yet effective mechanism to manage technical debt in the long term. The findings on the second research question revealed that projects where code quality is a frequent topic in board meetings, are having higher chances to reduce their TD density through the improvement of new code. The existence of explicit commit guidelines was not found to be significantly associated with the frequency of cleaner commits at the project level. However, it is noteworthy that several projects express explicit concerns about code quality in the commit guidelines offered to potential contributors. We list representative guidelines associated with code quality posted in the projects’ websites in Table 4.6. Such guidelines from well-known Apache projects can be considered as best practices and thus reused in other projects.

Ensuring that new code commits are as TD-free as possible, is a promising way of sustaining quality and avoiding quality degradation over time. Putting a Quality Gate in place is a relatively low-cost approach that sets an easy-to-manage, everyday goal for software developers, explicitly emphasizing code quality.

4.5.3 Implications for Researchers

With respect to researchers working in the area of Technical Debt Management the obtained evidence opens up further opportunities for studies on the impact of new code on quality. Research can focus on establishing whether systems with a degrading quality over time are associated with high $TD_{density}$ of new code and vice-versa. Given that no project exhibits a monotonous trend in its quality over time, it would be reasonable to perform such studies after splitting the timeline of system history into periods with monotonous and statistically significant trends in their quality in terms of $TD_{density}$. Declaring that clean new code results in improving quality over time would emit an explicit message to software developers.

Furthermore, the use of refactoring miners can be exploited to investigate which strategy (i.e., writing new code vs. applying regular refactorings) is more efficient for managing technical debt. Evidence on this central question would be highly relevant to software development teams considering both the effort that is associated with each type of quality-improvement approach as well as the attractiveness of each coding activity to developers. Moreover, it would be equally interesting to assess the quality of new code commits pertaining to specific change types, classified from the view point of the goal of change. Such classifications consider for example changes due to fault-fixing, feature addition, enhancements or general maintenance. (Hassan 2009, Palomba et al. 2017). Another research direction worth of investigation would be the study of ‘ripple’ effects of new code to the rest of the system.

Further studies that compare projects that rely on tools such as SonarQube and others that do not, could reveal whether the use of such platforms leads to TD reduction. Moreover, it makes sense to investigate whether there is any relation between the experience of developers and the quality of new code that they introduce. Apart from the analysis of software artifacts we believe that it would be highly interesting to reach out to management boards of open source or industrial projects to obtain their own perception on Technical Debt in new code and analyze their strategies for preventing its accumulation.

4.6 Threats To Validity

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

4.6.1 Construct Validity

Construct validity reflects how far the examined phenomenon is connected to the intended studied objectives. The main involved threat is related to the accuracy by which technical debt can be captured by static analysis tools such as SonarQube. Rule violations reported as Technical Debt Issues are obviously only one manifestation of actual code and design inefficiencies. The lack of any ground truth in technical debt measurement means that the accuracy of SonarQube, or any other technical debt tool, can hardly be validated. According to Martini et al. (Martini et al. 2018), currently static analyzers (such as SonarQube) are used in industry to analyze the source code in search of technical debt. Only in few cases out of the respondents in their survey (15 companies) practitioners built their own metrics tools for checking (language-specific) rules or patterns that can warn the developers of the presence of technical debt. In a similar discussion, Yli-Huuma et al. (Yli-Huuma et al. 2016)

found SonarQube to be the most used tool for TDM in the eight development teams that were involved in their case study.

Furthermore, it is known that such tools are not capable of identifying architectural problems or other types of technical debt such as requirements, documentation, test or build debt. This threat however, is partially mitigated by the fact that SonarQube is one of the most widely used tools for technical debt identification and quantification. Moreover, we have used the same tool both for assessing the system's $TD_{density}$ as well as the contribution of new / deleted / modified code on the $TD_{density}$ change. Furthermore, the analysis of $TD_{density}$ evolution is based mostly on a relative assessment of the changes rendering the measurement of absolute technical debt values less important for this type of study.

In addition, we investigate the effect of code insertion, deletion and modification on changes in the system's $TD_{density}$. Generally, these three types correspond to all possible changes in terms of code. However, we consider only Technical Debt that can be mapped to methods, thus ignoring TD which might occur at the level of classes or files. SonarQube reports violations at the class/file level; however, tracking the types of changes that can introduce or remove such violations requires a different study set-up. More importantly, we consider the co-occurrence of changes (e.g. an increase of $TD_{density}$ by new code and an increase in the system's $TD_{density}$). Thus, we do not capture the underlying causes of variation in the number of Technical Debt issues (such as the introduction of a new library or framework, the implementation of new functionality, etc.).

4.6.2 Reliability

Reliability reflects whether the study has been conducted and reported in a way so that others can replicate it and reach the same results. To mitigate this threat, the study protocol is extensively described in Section 4.3 explicitly listing all data collection and analysis steps. It should be emphasized that data has been subject only to automated analysis with no subjective interpretation from the researchers; therefore, researcher bias has been avoided. A replication package is available with all available data to allow for an independent replication of the investigation. Replication to other software ecosystems like the ones by Google, Android and Salesforce is deemed particularly important for validating the findings on the effectiveness of clean new code.

4.6.3 External Validity

External validity examines the applicability of the findings in other settings, e.g. other software projects, other programming languages and possibly other technical debt identification tools. We have focused only on Java open-source software projects that use maven as a build tool. This limits the ability to generalize the findings to other projects. The fact that the study focuses on twenty-seven projects of the Apache Software Foundation which are highly

active and popular among software developers partially mitigates threats to generalization. Nevertheless, the focus on a specific software ecosystem is still a source of threat to the generalizability of the results. Last but not least, the introduction of TD through new code has been analyzed in this study only on open source projects. In industry, projects are characterized by a very tight schedule; this means that our findings cannot be generalized to industrial systems. Thus, replication studies on the effect of new code on the evolution of technical debt are needed to strengthen the validity of the derived conclusions to industrial systems or systems that use different programming languages and environments.

4.7 Conclusions

The Technical Debt metaphor is usually associated with degrading quality trends in software systems. However, not all systems *age* over time. In this paper we have made an attempt to shed light into the drivers of technical debt change, by focusing on the contribution of new, deleted and modified code. In particular, we have performed an empirical study on the entire history of twenty-seven open-source projects by analyzing the changes, at method level, for each individual commit. By mapping Technical Debt Issues to new, deleted, and modified methods we have been able to investigate the association between the types of changes and the variation in the system's $TD_{density}$.

The results revealed that the quality of new code in terms of its $TD_{density}$, is, for the majority of the revisions, higher than the quality of the system in which the code is introduced. Moreover, among new, deleted, and modified code, the contribution of code modification exhibits a strong association to the change in the system's $TD_{density}$, followed by the contribution of new code. The contribution of new code is more profound for the transitions in which the quality of the system improved. More specifically, it was found that adding new code that is *cleaner* than the existing codebase coincides very frequently with a reduction in $TD_{density}$ of the system. The same association, has also been observed for the contribution of method modification to the change of the total $TD_{density}$. Finally, we have found indications that projects in which code quality is often discussed in their board meetings, exhibit a higher frequency of cleaner code commits.

The findings of this study suggest that writing code that has fewer Technical Debt Issues than the host code, can prove a very efficient and low-cost approach for managing TD. Applying Quality Gates to ensure that each commit yields fewer violations than the average, essentially leads to an improving quality trend, thereby reversing the software ageing phenomenon. Further studies can reveal whether writing clean new code offers a better cost/benefit ratio than the widely studied strategy of software refactoring.

Table 4.3: Relation between contribution of new/deleted/modified methods and change in system's $TD_{density}$ (RQ₁)

$TD_{density}$ Change		Contribution to $TD_{density}$ per new/delete/modified methods								
		New			Deleted			Modified		
		↓	↑	-	↓	↑	-	↓	↑	-
Accumulo	↓	195	47	0	51	52	1	205	39	18
	↑	97	113	0	15	66	0	55	163	5
Atlas	↓	446	56	1	73	156	2	373	126	25
	↑	141	153	2	31	102	2	62	249	7
Beam	↓	550	86	8	137	235	11	542	131	33
	↑	172	198	8	44	184	8	90	311	17
Calcite	↓	607	27	0	44	157	3	446	165	32
	↑	209	108	0	23	102	5	53	277	1
Cayenne	↓	528	46	2	88	202	2	421	135	58
	↑	173	111	1	22	155	4	52	269	12
CXF	↓	119	109	7	90	285	7	917	262	97
	↑	344	310	3	35	178	4	142	548	15
DeltaSpike	↓	102	18	1	21	24	2	95	15	15
	↑	39	36	1	5	23	3	25	55	2
Drill	↓	521	51	1	77	181	6	427	118	35
	↑	212	78	1	17	104	4	23	274	3
Dubbo	↓	144	15	1	28	62	2	123	32	12
	↑	54	27	3	5	35	4	10	80	0
Flink	↓	1560	210	9	307	673	20	1314	465	76
	↑	477	402	5	126	416	13	222	756	18
Flume	↓	224	25	2	21	67	1	178	59	16
	↑	53	69	0	11	33	0	29	99	2
Giraph	↓	172	29	1	31	67	2	143	48	9
	↑	56	47	2	5	33	1	19	91	1
Jackrabbit	↓	915	74	1	111	298	7	711	245	89
	↑	212	170	6	35	149	4	86	345	16
jclouds	↓	1109	66	7	136	433	14	821	294	131
	↑	272	200	6	29	268	6	105	429	24
Knox	↓	271	35	21	44	68	2	241	74	24
	↑	68	88	7	23	44	5	55	124	5
Kylin	↓	714	127	1	177	295	4	688	174	47
	↑	255	254	4	60	211	7	116	460	16
Metron	↓	206	21	1	30	76	1	147	72	9
	↑	71	30	0	15	31	0	13	92	0
MyFaces	↓	205	17	0	37	57	3	176	56	21
	↑	65	96	0	11	44	3	37	138	6
NiFi	↓	473	32	2	50	113	3	337	144	36
	↑	136	97	0	16	67	2	55	188	2
oozie	↓	131	22	1	30	58	0	148	51	13
	↑	60	39	0	5	22	0	21	83	1
OpenWebBeans	↓	330	38	8	71	107	3	308	88	31
	↑	95	74	5	17	76	5	39	167	5
PDFBox	↓	385	38	1	60	85	1	345	66	46
	↑	107	110	0	13	68	1	56	172	11
Pulsar	↓	427	54	1	44	100	4	350	117	33
	↑	117	140	4	24	78	0	63	214	8
SIS	↓	205	22	1	52	98	1	152	74	10
	↑	81	94	1	31	84	2	44	144	2
Storm	↓	133	28	2	37	47	4	136	25	8
	↑	61	86	1	30	47	0	40	116	1
TinkerPop	↓	1170	151	2	200	477	23	949	299	145
	↑	414	393	5	80	350	25	174	696	25
Zeppelin	↓	368	57	0	68	91	3	332	93	27
	↑	155	137	1	22	63	3	61	236	9

Table 4.4: Chi-squared test for New, Deleted, and Modified methods

Project		New	Deleted	Modified
Accumulo	χ^2	61.55**	20.32**	174.75**
	ϕ	0.368	0.324	0.595
Atlas	χ^2	227.08**	75.86**	238.93**
	ϕ	0.532	0.455	0.532
Beam	χ^2	252.51**	32.53**	371.67**
	ϕ	0.495	0.227	0.572
Calcite	χ^2	155.09**	2.49	301.03**
	ϕ	0.403	0.086	0.555
Cayenne	χ^2	154.05**	29.40**	306.58**
	ϕ	0.421	0.247	0.565
CXF	χ^2	499.40**	18.52*	618.96**
	ϕ	0.512	0.174	0.555
DeltaSpike	χ^2	46.84**	9.18	68.50**
	ϕ	0.484	0.337	0.568
Drill	χ^2	50.00**	11.66	405.69**
	ϕ	0.240	0.173	0.677
Dubbo	χ^2	24.30**	9.21	118.01*
	ϕ	0.315	0.258	0.675
Flink	χ^2	425.38**	19.24*	686.37**
	ϕ	0.399	0.111	0.488
Flume	χ^2	95.41**	0.52	100.28**
	ϕ	0.505	0.062	0.511
Giraph	χ^2	74.01**	13.22	106.43**
	ϕ	0.489	0.304	0.582
Jackrabbit	χ^2	289.10**	10.65	380.25**
	ϕ	0.457	0.132	0.502
jclouds	χ^2	486.34**	30.43**	464.59**
	ϕ	0.539	0.184	0.504
Knox	χ^2	154.25**	42.84**	165.54**
	ϕ	0.559	0.475	0.559
Kylin	χ^2	193.14**	45.06**	512.66**
	ϕ	0.377	0.243	0.582
Metron	χ^2	22.99**	0.72	91.61**
	ϕ	0.264	0.069	0.524
MyFaces	χ^2	198.44**	10.18	129.66**
	ϕ	0.715	0.251	0.541
NiFi	χ^2	360.68**	38.11**	189.47**
	ϕ	0.695	0.387	0.496
oozie	χ^2	34.55**	2.81	87.28**
	ϕ	0.337	0.156	0.523
OpenWebBeans	χ^2	97.28**	21.01**	208.81**
	ϕ	0.420	0.271	0.568
PDFBox	χ^2	237.85**	27.78**	231.95**
	ϕ	0.605	0.344	0.572
Pulsar	χ^2	288.13**	16.38*	212.82**
	ϕ	0.622	0.255	0.519
SIS	χ^2	227.86**	35.03**	122.02**
	ϕ	0.750	0.360	0.533
Storm	χ^2	133.34**	4.79	117.20**
	ϕ	0.654	0.170	0.599
TinkerPop	χ^2	480.66**	30.52**	706.49**
	ϕ	0.473	0.161	0.551
Zeppelin	χ^2	100.30**	9.87	240.53**
	ϕ	0.374	0.197	0.561

** p < 0.01, * p < 0.05

Table 4.5: Adoption of Commit Guidelines and frequency of references to code quality issues in board meetings per project

Project	Commit Guidelines	Project Board Meetings
Accumulo	NO	LOW
Atlas	NO	LOW
Beam	NO	HIGH
Calcite	YES	HIGH
Cayenne	NO	LOW
CXF	YES	LOW
DeltaSpike	YES	LOW
Drill	YES	HIGH
Dubbo	YES	LOW
Flink	YES	LOW
Flume	NO	LOW
Giraph	NO	LOW
Jackrabbit	NO	HIGH
jclouds	YES	LOW
Knox	NO	LOW
Kylin	YES	LOW
Metron	YES	HIGH
MyFaces	YES	LOW
NiFi	YES	HIGH
oozie	NO	LOW
OpenWebBeans	NO	HIGH
PDFBox	YES	HIGH
Pulsar	YES	LOW
SIS	NO	HIGH
Storm	YES	LOW
TinkerPop	YES	LOW
Zeppelin	YES	LOW

Table 4.6: Representative Commit Guidelines

Project	Representative Commit Guidelines
Calcite	<i>Trigger a Coverity scan ... and when it completes, make sure that there are no important issues.</i>
CXF	<i>Make use of both PMD and CheckStyle to enforce common coding conventions.</i>
DeltaSpike	<i>Follow project's formatting rules and always build and test your changes before you make pull requests</i>
Drill	<i>Code should be formatted according to Sun's conventions, contributions should not introduce new Checkstyle violations, contributions should pass existing unit tests, and new unit tests should be provided to demonstrate bugs and fixes</i>
Dubbo	<i>Dubbo uses code style that is almost in line with the standard java conventions and suggests the contributors to implement a few unit tests for a new feature or an important bugfix.</i>
Flink	<i>Flink i) suggests the developers to comment as much as necessary to support code understanding, ii) provides guidelines for good design and software structure, iii) guidelines for good concurrency and threading and iv) suggestions for dependencies and modules</i>
jclouds	<i>Contexts and APIs are thread-safe (or should! Otherwise it is an issue)</i>
Knox	<i>Adding new service API support, the committer should give sufficient tests and documentation</i>
Kylin	<i>The changes MUST be covered by a unit test or the integration test, otherwise it is not maintainable</i>
Metron	<i>Try-finally used as necessary to restore consistent state, Appropriate NullPointerException and IllegalArgumentException argument checks</i>
MyFaces	<i>Error and exception handling: If the exception is severe, but there is a chance to continue processing, a message with severity "error" or "warning" should be logged..</i>
NiFi	<i>If an unexpected RuntimeException is thrown, it is likely a bug and allowing the framework to rollback the session will ensure no data loss</i>
PDFBox	<i>The new code should follow the project's coding conventions where possible.</i>
Pulsar	<i>All code should have appropriate unit testing coverage. New code should have new tests in the same contribution. Bug fixes should include a regression test to prevent the issue from reoccurring.</i>
Storm	<i>The most important is consistently writing a clear docstring for functions, explaining the return value and arguments. As of this writing, the Storm codebase would benefit from various style improvements..</i>
TinkerPop	<i>A rich set of algorithms is an important goal for MLLib, scaling the project requires that maintainability, consistency, and code quality come first.</i>
Zeppelin	<i>The project follows Google's Java Code style and suggests the developers to use some formatting plugins to lint their code</i>

Published as:

G. Digkas, A. Ampatzoglou, A. Chatzigeorgiou, and, P. Avgeriou “The Temporality of Technical Debt Introduction on New Code and Confounding Factors,” *Software Quality Journal* , doi: 10.1007/s11219-021-09569-8.

Chapter 5

The Temporality of Technical Debt Introduction on New Code and Confounding Factors

Abstract

Code Technical Debt (TD) is intentionally or unintentionally created when developers introduce inefficiencies in the codebase. This can be attributed to various reasons such as heavy workload, tight delivery schedule, or developers' lack of experience. Since a software system grows mostly through the addition of new code, it is interesting to study how TD fluctuates along this process. Specifically, in this paper we investigate: (a) the temporality of code TD introduction in new code, i.e., whether the introduction of TD is stable across the lifespan of the project, or if its evolution presents spikes; and (b) the relation of TD introduction to the development team's workload in a given period, as well as to the experience of the development team. To answer these questions, we have performed a case study on 47 open source projects from two well-known ecosystems (Apache and Eclipse) as well as additional isolated projects from GitHub (not selected from a specific ecosystem) and inspected the number of TD issues introduced in 6-month sliding temporal windows. The results of the study suggested that: (a) overall, the number of TD issues introduced through new code is a stable measure, although it presents spikes; and (b) the number of commits performed, as well as developers' experience are not strongly correlated to the number of introduced TD issues.

5.1 Introduction

Technical Debt (TD) at the code level refers to inefficiencies introduced in the source code of an application during the implementation or the maintenance phase (Li et al. 2015). These inefficiencies manifest themselves as violations of coding standards, complex and hard to understand code, code duplicates, etc. (Letouzey 2012). According to Alves et al. (Alves et al. 2010) code technical debt is the most studied type of technical debt, and based on Ampatzoglou et al. (Ampatzoglou et al. 2016) it is one of the most important in industry.

One of the certainties in software development is continuous change (Lehman 1996), and code technical debt is no exception to this. In fact, there has been significant work on how

code technical debt evolves and how it accumulates over time. However, existing studies have looked at technical debt evolution as a whole, without distinguishing between technical debt that is added as new code, and technical debt incurred through the modification of existing code. In this paper, we focus only on the introduction of new code technical debt, i.e., TD inserted in the system in the form of new Technical Debt issues. More specifically we study new methods (our scope is object-oriented systems) that contain technical debt and we look at the introduction of new technical debt as a temporal phenomenon. This can shed light on whether technical debt is introduced in an almost flat rate, whether large volumes of technical debt are introduced in infrequent timestamps, or whether large volumes of TD are frequently introduced along evolution; examples of these three cases are presented in Figure 5.1. Consequently, in the second and third case, we need to revise our technical debt management techniques to focus on those times where large TD is introduced to prevent or limit the phenomenon. Broadly speaking, the introduction of TD items in new code can either depend on the capabilities of the development team or be associated with external factors such as feature requests in short time, lack of sufficient time for testing, etc. The former would be reflected in a uniform introduction of TD along evolution, whereas the latter would be recognized by fluctuations in the introduction of TD. Knowing the particular circumstances can help towards self-improvement of the development process so as to address the root causes of TD accumulation.

Figure 5.1: TD Introduction Temporality

Focusing on technical debt that is introduced by new code, as opposed to technical debt that is introduced by modifying existing code, can provide a unique insight. Specifically, the new Technical Debt issues introduced by new methods at each commit (either new methods in existing classes or new methods in entirely new classes) reflect more accurately the type of problems and the timepoint at which they are introduced. In other words, new methods are more representative of the developers' practices and knowledge level, compared to method modifications whose type and timeliness is often dictated by the need to fix a bug or to extend an already existing functionality. Thus, we study the temporality of technical debt through a

clearer source. Furthermore, in our previous work we established that through the systematic introduction of ‘clean’ new code, a software system can gradually be freed of its technical debt (Digkas, Chatzigeorgiou, Ampatzoglou and Avgeriou 2020).

In this paper, we answer two high-level questions: (Q1) if the number of introduced Technical Debt issues is uniformly spread across evolution, or whether there are time windows in which more Technical Debt issues are inserted; and (Q2) if the number of Technical Debt issues that is introduced along evolution is related either to the activity (intensity of commits) of developers in different time windows, or to the experience of the developers. Projects could exhibit either a stability in the introduction of code Technical Debt issues across evolution or experience fluctuations with isolated or repeating spikes of introduced code Technical Debt issues. In the former case one could assume that accumulation of technical debt is most probably due to factors that are constantly present in the entire lifetime of the project, such as used processes, tools, management practices, etc. In the latter case, one could postulate that the insertion of new code Technical Debt issues is a highly temporal phenomenon depending on volatile factors such as feature requests, changing schedules, pressure to fix bugs, or human factors (such as team composition, developers’ involvement and experience).

To achieve this goal, we explore the evolution of 47 open source projects from two well-known ecosystems, namely: Apache Software Foundation (ASF) and Eclipse Foundation (EF). However, since ecosystems exhibit particular habits and regulations, we have included additional OSS projects from GitHub to increase the generalizability of the findings. In particular, we track the number of new Technical Debt issues inserted in each commit. Next, we create a 6-month sliding window (the duration of the time window has been set to 6 months as in the study by Hassan on predicting faults using the complexity of code changes (Hassan 2009)), and we calculate the cumulative number of inserted Technical Debt issues for each window, as well as the number of commits in the same window and the weighted (by the contribution of each developer) average experience of the development team. To answer the first question (Q1), we use a metric property (termed SMF—see Section 5.3) that is able to assess metrics fluctuation along time and characterize them as either stable or sensitive. To answer the second question (Q2) we correlate the number of inserted Technical Debt issues with a) the number of commits for each window and b) the weighted experience of the development team. The reporting and interpretation of the results is performed at the project level.

This study is an extended and revised version of our previous work (Digkas, Ampatzoglou, Chatzigeorgiou and Avgeriou 2020) in which we also explored the temporality of Technical Debt issues introduction. The main differences of this work compared to the previous one, are as follows:

- **Sample size.** It explores a significantly larger dataset of 47 instead of 27 projects, resulting into the exploration of (approx.) 2.5K instead of (approx.) 2K time windows.

- **Generalizability.** It strengthens the generalizability of our results, since it performs the analysis on two software ecosystems (Apache and Eclipse projects), compared to only one. In addition, we now explore isolated projects, i.e., not related to a specific ecosystem.
- **Accuracy of data analysis.** The analysis of spikes along evolution (i.e., the timestamps with large introductions of technical debt amounts) is performed through statistical analysis (extreme values detection), instead of visual observation.
- **Exploration of additional evolution parameters.** We explore also the relation between developers' experience and Technical Debt issues introduction through an additional research question. In the original paper, we had only looked into the relation between Technical Debt issues introduction and development effort intensity.

The rest of the paper is organized as follows: in Section 5.2 we present related work and in Section 5.3 background information important for understanding the study. In Section 5.4, we present the design of the case study, while Section 5.5 elaborates on the results. Section 5.6 interprets the results and provides implications for researchers and practitioners. Finally, in Section 5.7 we present threats to validity and in Section 5.8, we conclude the paper.

5.2 Related Work

Since this paper focuses on the introduction of technical debt over time, we organize this sub-section into causes of technical debt introduction (see Section 5.2.1) and technical debt evolution (see Section 5.2.2).

5.2.1 Causes of Technical Debt Introduction

Quite recently the research consortium of the InsignTD project¹ has explored the causes and effects of technical debt accumulation in various countries along the globe (e.g., Brazil, Chile, Colombia, Serbia, USA, etc.) through a family of surveys (Pérez et al. 2019, Ramac et al. 2020, Rios et al. 2018). Through these research efforts they have identified in total 78 causes and 66 effects. Among the identified causes of technical debt, close deadline, inappropriate planning, and lack of knowledge/experience have proven to be the most prolific ones. Additionally, Tufano et al. (Tufano et al. 2017) studied the evolution of code smells with the goal of understanding when and why code smells are introduced and observed the life cycle of five code smells. The results indicate that: (a) in the majority of the cases code smells are introduced with the creation of the corresponding classes or files; (b) while

¹<http://www.td-survey.com/>

projects evolve, “smelly” code artifacts tend to become more problematic; (c) new code smells are introduced when software engineers implement new features or when they extend the functionality of the existing ones; (d) the developers who introduce new code smells, are the ones who work under pressure and not necessarily the newcomers; and (e) the majority of the smells are not removed during the project’s evolution and few of them are removed as a direct consequence of refactoring operations.

According to Kazman et al. (Kazman et al. 2015) who conducted a case study on the roots of architecture debt (ATD), is extremely common and probably the most important type of technical debt because it consumes the largest percentage of maintenance effort. Their findings suggest that architectural debt is extremely easy to introduce: programmers typically want to introduce new features or fix bugs; however, by changing the code they often undermine the architectural structure leading to the accumulation of ATD. Finally, Martini et al. (Martini et al. 2015) conducted a case study on five software companies to understand the causes that introduce ATD. Large software companies try to deliver as fast as possible in order to satisfy their customers’ needs, usually taking shortcuts, thereby introducing ATD. If the debt is not paid-off, it starts to accumulate and this makes feature development more difficult. However, we clarify that the current study does not deal with ATD but rather focuses on code-level TD.

5.2.2 Evolution of Technical Debt

Although technical debt is a multifaceted concept, one of the key constituents of code technical debt is the presence of code smells. One of the first studies that investigate the evolution of code smells was conducted by Olbrich et al. (Olbrich et al. 2009). They investigated the evolution of two code smells, God Class and Shotgun Surgery, on two OSS projects. The results show that along software development, there are phases where the number of code smells can either increase or decrease and those phases are not affected by the size of the systems. Chatzigeorgiou and Manakos (Chatzigeorgiou and Manakos 2014) have investigated the evolution of the Long Method, Feature Envy, State Checking, and God Class smells in two open-source software projects. The results suggested that as projects evolve the number of smells tends to increase. Another interesting finding is that a significant percentage of smells was not due to software ageing, since some smells were present right from the first version of the code in which they reside. Peters and Zaidman (Peters and Zaidman 2012) studied the lifespan of the God Class, Feature Envy, Data Class, Message Chain Class, and Long Parameter List smells. The analysis of eight open-source software projects, confirmed that the number of smells increases, as projects evolve.

Digkas et al. (Digkas et al. 2017) tracked the evolution of technical debt in sixty-six open-source Java projects by the Apache Software Foundation, over a period of 5 years. In order to detect issues that incur technical debt, they relied on SonarQube. The results show

that on the one hand, there is a significant increasing trend on the size, complexity, number of Technical Debt issues, and the total technical debt over time, which seems to confirm the software aging phenomenon. But on the other hand, when technical debt is normalized over the non-commented lines of code, an evident decreasing trend over time is present for many of the projects. This could possibly be attributed to: (a) developers that perform refactoring activities and fix some of the open Technical Debt issues; or (b) developers that introduce better quality code in each commit (compared to the project’s existing code base). We summarize the key findings of the related work in Table 5.1.

Table 5.1: Causes and Evolutionary Patterns of TD/smells

Causes	Study
Close deadlines, inappropriate planning, lack of knowledge/experience	(Pérez et al. 2019, Ramac et al. 2020, Rios et al. 2018)
Pressure	(Tufano et al. 2017)
Architecture flaws	(Kazman et al. 2015)
Pressure to deliver	(Martini et al. 2015)
Evolutionary Patterns	Study
Smells tend to increase and decrease in different phases	(Olbrich et al. 2009)
Smells increase monotonically	(Chatzigeorgiou and Manakos 2014)
Smells increase monotonically	(Peters and Zaidman 2012)
TD increases monotonically (Normalized)	(Digkas et al. 2017)
TD decreases for some projects)	

5.3 Background Information

Prior to the presentation of the case study design and results we discuss background material related to the identification of technical debt in new code and the fluctuation of software metrics. While more details can be found on the references we provide, here we present an overview that is necessary to understand our data collection and analysis and keep this publication self-contained.

Identifying Technical Debt issues on new code. SonarQube is one of the most widely used tools for assessing the level of code technical debt present in a software system (Amanatidis et al. 2020, Digkas, Chatzigeorgiou, Ampatzoglou and Avgeriou 2020). According to a recent overview of TD tools (Avgeriou et al. 2020) SonarQube is by far the most popular

tool based on its trace in the scientific literature and online media channels. We have relied on version 7.9.2 to identify and quantify technical debt in individual commits throughout the history of the examined projects. SonarQube estimates technical debt principal using a set of predefined rules² such as *Methods should not be too complex*, *Inheritance tree of classes should not be too deep*, or *Classes should not be coupled to too many other classes*. Each identified issue is assigned a remediation cost corresponding to the time required to fix the corresponding rule violation. The sum of the remediation costs for all issues yields the reported TD principal.

As *new code* we consider the new methods which are introduced in each commit. Such methods can be added either in existing classes or in entirely new classes. We do not consider new code in the form of new instructions in existing methods (i.e. modifying methods). The reason is that, beyond the complexity of tracking changes at the instruction level, an entirely new method conveys better the programming habits of a developer as it refers to a complete, self-standing piece of functionality. To distinguish the newly inserted methods for each commit from the deleted, modified, renamed, and unchanged methods, we rely on the Gumtree Spoon AST Diff tool (Falleri et al. 2014). Our approach identifies all changes per commit at the file-level, i.e. we detect files which have been added, modified, renamed, and deleted. For the added files or classes, we consider all of their methods as new. For the modified and renamed files we compare their representation in the form of an Abstract Syntax Tree with the one of the previous revision; subsequently we identify the newly inserted methods in existing classes.

We then proceed with identifying all Technical Debt issues in newly added methods, by running a technical debt analysis with SonarQube. The identified issues are obtained through SonarQube’s API from which we retain only Technical Debt issues found within the line range of new methods.

Assessing the fluctuation of Technical Debt issues. Observing the evolution of a metric value throughout the history of a software project can be considered as the analysis of a time series. The time series can exhibit volatility depending on the metric itself and on the nature of changes throughout the history. In previous work, we have defined Software Metrics Fluctuation (SMF) as “*the degree to which a metric score changes from one version of the system to the other*” (Arvanitou et al. 2016). According to the SMF property a metrics can be characterized as *sensitive* (changes induce high variation on the metric score) or *stable* (changes induce low variation).

As this study focuses on the evolution of the quality of new code, in terms of its TD, we make use of the SMF property. In particular we employ the mf measure (Arvanitou et al. 2016) which is defined as: “*the average deviation from zero of the difference ratios between every pair of successive versions*”. We adapt the mf measure for the case of Technical Debt issues, resulting in formula (5.1). $TDissues(i)$ refers to the number of Technical Debt issues

²<https://rules.sonarsource.com/>

identified at version i of the history, while n is the total number of analyzed time snapshots (e.g. versions). A zero deviation of the number of Technical Debt issues from that of the previous version implies no fluctuation. The squared root of the second power of the ratio of the difference between any two successive versions yields a measure similar to the standard deviation. The closer to zero mf gets, the more stable the number of Technical Debt issues is.

$$mf = \sqrt{\frac{\sum_{i=2}^n \left(\frac{TDissues_i - TDissues_{i-1}}{TDissues_{i-1}} \right)^2}{n - 1}} \quad (5.1)$$

5.4 Case Study Design

Case study is an observational method that is used for studying a phenomenon in its real-life context. In our study, the phenomenon is the temporality of TD in new code, while the context refers to the evolution of open-source software projects. In this section, we present the design of the case study, organized based on the linear-analytic structure as described by Runeson et al. (Runeson et al. 2012).

5.4.1 Research Questions

The high-level goal of this study is the identification of temporal patterns in the introduction of code technical debt, along software evolution. To explore this goal, we need to investigate two sub-goals. First, we look at the fluctuation of the number of Technical Debt issues introduced by new code along the evolution of software projects. Second, we explore the relation between the number of Technical Debt issues introduced by new code and two factors: (i) the number of commits that occurred in the same period; and (ii) the developers' experience. These sub-goals lead to the following two research questions.

RQ₁: Does the number of code Technical Debt Issues introduced by new code fluctuate along evolution?

The answer to this research question will unveil if in different time periods, different amounts of technical debt are introduced. The answer reflects the main goal of this study, i.e., to investigate the temporality of the technical debt phenomenon. Specifically, this answer will enable us to characterize Technical Debt issues introduction as either stable, or sensitive to temporal influence. In addition, we will study any possible spikes in the evolution of new code technical debt, which might be indicators of “extra-ordinary” events along evolution. The frequency and the timing (early, middle, or late in the project) of such spikes will also be explored and reported.

RQ₂: Does the amount of code Technical Debt introduced by new code correlate to the activity or the experience of the developers?

To increase the confidence in the results of the previous research question, we study two potentially important confounding factors for this empirical setup: i.e., developers' activity (RQ2.1) and developers' experience (RQ2.2). Considering that we are not analyzing at the individual commit level, but over periods of time, there is a non-negligible chance that in these periods the developers' activity (number of commits) is not stable; therefore, spikes in new code Technical Debt issues could be due to more intense programming activity in the corresponding periods. Additionally, we explore if the experience of a team as an aggregation of the experience of the team members plays a role in the introduction of TD; in other words, we explore if the technical debt introduction in projects correlates with the experience of its team members.

5.4.2 Cases and Units of Analysis

This study is characterized as a multiple, embedded case study (Runeson et al. 2012), in which the cases are open-source software (OSS) projects, while the units of analysis are the source code commits (per project) over different time periods. Specifically, for each project, we analyse the number of code Technical Debt issues added over 6-month time periods across the project history (see Section 5.4.3 for more details). The reason for selecting to perform this study on open-source software systems is the vast amount of data that is available in terms of revisions and classes. The long history that is available for each project enables researchers to observe overall trends in the evolution of their quality. To retrieve data from high-quality projects that evolve over a period of time, we looked into ASF and EF projects (Dueñas et al. 2007) as well as additional OSS projects and investigated the projects presented in Table 5.2. The demographics of Table 5.2 include the number of classes, number of non-commented lines of code, the number of analyzed revisions, the date of the first commit, number of issues in the corresponding GitHub issue tracker as well as the number of developers per project. The two ecosystems have been selected since they are well-known in the software engineering community for their quality, and structured development processes. On the one hand, Eclipse is an industry-driven initiative involving around 100 companies, universities, and contributors who deliver OSS based on the Eclipse environment. On the other hand, Apache Software Foundation is a highly successful initiative, made up of individuals, that provides popular, high quality, OSS—providing support for over half of the world's websites. Both communities follow a mature approach in developing software, having established processes to face the problems inherent to software development. However, to reduce generalizability threats from the study of projects belonging to well known and systematically maintained ecosystems, we considered twelve additional OSS projects.

The selection of projects was based on the following criteria:

Table 5.2: Selected Projects' Demographics

	Project	Classes	NCLOC	Revs	1 st Commit	Issues	Devs
Apache Software Foundation	Atlas	932	87637	1454	11/20/2014	134	104
	Beam	3757	176663	2780	12/13/2014	14733	763
	Calcite	2606	186633	1448	04/20/2012	2408	253
	Cayenne	2615	164170	2116	01/21/2007	449	34
	Commons IO	132	10500	1059	01/25/2002	224	71
	CXF	4111	353085	5079	04/23/2008	787	160
	DeltaSpike	951	46182	513	12/22/2011	114	57
	Drill	4655	316552	1316	09/03/2012	2216	163
	Dubbo	943	61865	728	10/20/2011	4230	386
	Flink	5632	341149	5329	12/15/2010	15837	870
	Flume	790	51897	789	08/02/2011	333	54
	Giraph	1414	72972	668	10/29/2010	149	26
	Jackrabbit	2883	273574	4260	09/13/2004	73	23
	jclouds	5687	227459	4323	04/11/2009	94	237
	Knox	1083	51429	1033	10/24/2012	435	57
	Kylin	1658	128531	3205	05/13/2014	1637	187
	Metron	1433	72579	548	12/08/2015	1582	62
	MyFaces	1843	174158	1211	01/17/2006	200	30
	NiFi	4256	371031	1490	12/08/2014	5046	356
	oozie	1082	97597	587	08/19/2011	57	16
	OpenWebBeans	561	44299	1583	11/22/2008	32	18
	PDFBox	1279	136916	3758	02/10/2008	3326	6
	Pulsar	1837	147182	1503	09/07/2016	109	393
	SIS	1948	181588	828	03/20/2010	20	7
	Storm	3958	243574	738	09/16/2011	3396	341
	TinkerPop	1698	95652	5178	09/02/2013	1413	142
	Zeppelin	1209	89193	1562	06/19/2013	4109	344
Eclipse Foundation	Hono	554	43697	1977	02/08/2016	1076	38
	JGit	1314	100973	1747	09/29/2009	114	140
	JNoSQL	521	21285	1384	07/14/2016	111	15
	Kapua	2275	98005	1912	10/14/2016	1610	32
	Leshan	389	27378	808	02/07/2015	494	38
	Milo	1618	114581	261	05/06/2016	446	18
	Tycho	872	56552	290	09/19/2011	66	56
	Vorto	599	29011	1051	03/02/2015	1334	42
Other OSS	apollo	604	31642	2487	03/04/2016	2438	75
	gson	91	8085	1485	09/01/2008	1378	106
	hutool	1052	80819	1937	08/14/2019	1210	101
	jedis	157	28215	1801	06/11/2010	1384	175
	litemall	583	59588	1121	03/22/2018	324	50
	mybatis-3	414	21864	3726	05/17/2010	1075	176
	seata	1646	126362	1230	01/09/2019	2058	208
	spring-boot-admin	239	8964	40	06/04/2014	1338	117
	spring-cloud-alibaba	479	20855	1748	12/01/2017	1384	103
	webmagic	199	7970	1118	04/23/2013	843	40
	xxl-job	119	8666	1676	11/28/2015	2115	52
	zheng	484	30220	1241	10/04/2016	95	8

- The software is actively maintained. To ensure this, we sorted projects based on the date of their last commit.
- The software is written in Java and uses Maven as a build tool. This ensures that the project can be built and allows the retrieval of the project's language version from the corresponding pom.xml file.
- The software contains more than 100 classes to ensure the inclusion of systems with a substantial size, functionality and complexity.
- The software has more than 1000 commits. This criterion is used for similar reasons to the previous one, and to be able to observe trends in the evolution of their quality. Moreover, this number of revisions provides an adequate set of repeated measures as input to the statistical analysis.

5.4.3 Data Collection

To build the dataset for our analysis, we relied on the process described in Section 5.3. In particular, for each project, we have been able to build a dataset containing: (a) the commit SHA; (b) the committer; (c) the experience of the committer at the current timestamp; and (d) the number of introduced Technical Debt issues by the new code of this commit. Next, starting from the first commit timestamp, we created a 6-month time-window that slides monthly, along the evolution of the project. Based on these time-windows, we have created our units of analysis, as shown in Fig. 5.2. For example, by considering a project that spans across 22 months (M1-M22), we are able to create 16 units of analysis.

For each period captured by the time-window, we summed the number of Technical Debt issues that were introduced in all commits included in the time-frame. Therefore, the final data-set consists of four variables: $[V_1]$ time-window (in months/year); $[V_2]$ number of Technical Debt issues introduced by new code in the time-window; $[V_3]$ number of commits in the time-window; and $[V_4]$ the development team experience in the time-window.

With regards to the developers experience it should be highlighted that experience is not a directly observable construct (Siegmund et al. 2014). As a result, experience can be operationalized through multiple variables and be measured in different ways. For example, experience has been measured in terms of years of programming (Dieste et al. 2017), volume of commits (AlOmar et al. 2020), years of experience expressed in Likert-scale codes (Siegmund et al. 2014), etc.

In this study we need to capture the collective experience of an entire team combining both years of experience and volume of commits. The development team experience is a weighted average of developers' experience, over the contributions of a committer in the specific period, as explained in equations (5.2) - (5.4). In particular, first we calculate for each developer his/her experience in days at the beginning of each sliding window. This is

Figure 5.2: Demarcating Units of Analysis (sliding temporal windows)

obtained by measuring the time from their git registration according to eq. 5.2. Next, for each time window, we calculate the contribution rate of each developer as the number of commits that a developer has contributed to the project over the total commits in that time window as shown in eq. 5.3. The weighted development experience for each developer is obtained by multiplying the contribution rate with the experience in days. Finally, the entire team's experience is the sum of each developer's experience (eq. 5.4). To enable the easy replication of this study, a replication package is available online³.

$$Experience(days)_{dev_i} = CommitTimestamp - GitRegistration_{dev_i} \quad (5.2)$$

$$ContributionRate_{dev_i} = \frac{numberOfCommits_{dev_i}}{totalCommits} \quad (5.3)$$

$$TeamExperience = \sum_{i=1}^{developers} ContributionRate_{dev_i} * Experience_{dev_i} \quad (5.4)$$

5.4.4 Data Analysis

Data analysis was performed on the aforementioned raw dataset. To answer RQ₁, for each project, we first assess fluctuation by calculating SMF and basic descriptive statistics of the

³https://drive.google.com/drive/folders/1rWSUI_9ppUA0eUx3kFzNbSpP1PCrcnmX

dependent variable $[V_2]$. Next, to visualize extreme projects (the most stable and most sensitive), we use a line chart representing the evolution of Technical Debt issues introduced by new code. To identify spikes, along evolution we have used a standard method for identifying extreme outliers in SPSS, as defined in equations (5.5) or (5.6). As a final step, we explore, if these spikes are concentrated in the beginning, middle, or end of the project.

$$value > Quartile3 + 3 * InterquartileRange \quad (5.5)$$

$$value < Quartile1 - 3 * InterquartileRange \quad (5.6)$$

To answer RQ₂, we performed Spearman correlation analyses between variables $[V_2]$ and $[V_3]$, and between $[V_2]$ and $[V_4]$. The choice of a non-parametric test, was based on the fact that for some projects, the pre-conditions of parametric tests were not met. For extreme sensitive cases we visualize the relation through scatter-plots, and present the co-evolution of the number of commits and the number of Technical Debt issues in a single line chart. For extreme stable cases, we visualize the relation between the average development team experience and the number of Technical Debt issues in a single line chart.

5.5 Results

5.5.1 Fluctuation of TD introduction along evolution (RQ1)

In Table 5.3, we present the fluctuation analysis for the number of TD issues introduced by new code. Based on this, for 28 out of 47 projects the number of TD issues introduced by new code can be considered as stable (dark cells), whereas for the rest 19 as sensitive (light grey cells). For ASF, the percentage of stable projects is 55%, for EF 38%, and for isolated projects 83%; whereas the mean values do not differ in a statistically significant manner (ANOVA F: 1.811 and sig: 0.17). By comparing the SMF of the two ecosystems, we observe that the percentage of stable and sensitive projects is similar, whereas the one for isolated projects is substantially larger. A tentative explanation of this is the existence of specific development guidelines in large OSS ecosystems (like Apache or Eclipse), which limit the “Bazaar” effect of collaborative development in “random” open-source projects. To provide a visual insight on the discussed fluctuations, in Fig. 5.3, we present the evolution of one stable project, namely *Metron*, and a sensitive one, namely *SIS*. We note that even for the most “stable” projects, some spikes still exist; however, the spikes are small in height. To study the spikes in more depth, we first calculate their frequency – see Figure 5.4(a). Based on the bar chart, for 51% of the projects the spikes appear in less than 5% of

Figure 5.3: Indicative project evolution **Figure 5.4:** Project Fluctuations

the commits; whereas 23% of the projects produce fluctuations in more than 10% of their commits. We note that the classification of the projects in the three groups (i.e., [0% - 5%], [5% - 10%], and more than 10%) was data-driven. The spikes percentage for the analyzed projects was less than 15% for 46 out of 47 projects in our dataset. Based on this, we split equally the margin of values to three equal ranges [0% - 5%], [5% - 10%], and [10% - 15%]. Since the number of projects with spikes more than 15% was quite low, we have merged those to the last class.

Additionally, in Figure 5.4(b), we observe that fluctuations of TD are distributed across the entire project lifetime (as early, we consider the first third of the studied time windows, as middle the second third, and as late the most recent third). Fluctuations are reduced along evolution, and this decrease is statistically significant, based on the results of performing Friedmans' ANOVA ($F: 4.91$ and $\text{sig}: 0.008$). The aforementioned observations are first indications that these spikes might be relevant to the time period that they appeared, confirming the relation between Technical Debt issues introduction and project maturity. Nevertheless, this finding needs further investigation.

Figure 5.5: Activity vs. TD Introduction – Dubbo project

5.5.2 Correlation of Fluctuation vs. Activity (RQ2.1) and Experience (RQ2.2)

Developers’ Activity: One of the first tentative interpretations on the existence of high spikes as those presented in Fig. 5.3, would be that in the corresponding time windows, lots of code has been committed. To explore the existence of this confounding factor, in Table 5.4 we highlight with light-gray cell shading (in column Corr. Coef.) the cases in which the correlation is strong (>0.7 (Field 2013)) and at the same time statistically significant ($p < 0.05$). The findings suggest that only in 33% of the projects this correlation is strong for ASF, 42% for EF, and 66% for isolated projects. Based on the above, overall, only in 42% of the projects, the commit activity could explain the fluctuations in the number of TD issues that is added by new code. To visualize this result, we present the scatter plot and the evolution of both variables in a single line chart, in Figs. 5.5(a)-5.5(b) for Dubbo (i.e. a project with a high correlation), and in Figs. 5.6(a)-5.6(b) for PDFBox (i.e. a project with a low correlation). In the scatter plots, each dot represents a 6-month period, mapping the values of the two variables for which we seek correlation. For strong correlations, dots would be near to the central diagonal suggesting that a high number of TD issues is observed when the number of commits in a period is also high.

Developers’ Experience: Next, we replicate the same correlation process, focusing this time on developers’ experience. Similarly to Table 5.4, in Table 5.5, we present the correlation coefficient, along with the significance of the correlation between developers’ experience and the amount of TD issues introduced in new code. The results suggest that this relation is strong in 14% of the cases for ASF and 0% of the cases for EF, and 16% for isolated projects. This result implies that developers’ experience can be ‘blamed’ only for a limited number of cases of heavy TD introduction. Fig. ?? visualizes cases of strong positive and negative correlations. Finally, we note the sign of the correlation is negative in 66% of the cases, suggesting that experienced teams are producing less TD issues. However, 50% of strong

Figure 5.6: Activity vs. TD Introduction – PDFBox project

Figure 5.7: Experience vs. TD Introduction – Left: Dubbo, Right: TinkerPop

correlations are positive, implying that the cases in which experienced teams are introducing heavy technical debt are not negligible.

5.6 Discussion

5.6.1 Interpretation of Results

The high-level goal of this study was to investigate if the introduction of Technical Debt issues (by adding new code) is a temporal phenomenon, that diverges over time. Based on the findings, some temporality can be claimed only for a number of projects. In particular, based on the fluctuation of Technical Debt issues due to the introduction of new code (see Section 5.5.1), we can classify the projects in three categories through visual inspection of the evolution graphs: (a) stable projects without any temporality—i.e., negligible fluctuations (spikes in less than 5% of commits); (b) stable projects that are not sensitive, but some “extraordinary” spikes occur (spikes in 5% - 10% of commits); and (c) sensitive projects (spikes in more than 10% of commits).

Based on the findings of RQ1, we can claim that the introduction of Technical Debt issues due to the insertion of new code is independent of time, for more than half of the projects (36 out of 47). This can be interpreted as an indication of project maturity, in the sense that consistent quality is achieved throughout evolution. However, even for these stable projects, the absence of fluctuations does not necessarily imply the absence of any trend. For example, in Fig. 5.3 we can see that the evolution of project *Metron* does not exhibit any spikes; however, its trend is clearly a decreasing one.

On the other hand, for a subset of the analyzed projects (12 out of 47), the introduction of new code Technical Debt issues is a temporal phenomenon, since many spikes exist in their evolution. For these projects, the number of introduced Technical Debt issues in each period is not stable, and it is reasonable to assume that it is influenced by some external parameters. This can be interpreted as a consequence of the dynamic nature of software development environments, where many factors change along time. This finding is contradicting previous research on the evolution of smells (summarized in Table 5.1) implying that smells tend to increase monotonically over time. Consequently, it is important to study potential confounding factors that drive the accumulation of Technical Debt issues along the evolution of a software project. A starting point for such factors can be the causes of TD outlined in the studies of Table 5.1). To some extent, this has been addressed in RQ2.

The second research question lead to rather unexpected findings: i.e., the number of commits made in a time period as well as the experience of developers, are not correlated to the number of introduced Technical Debt issues into the system (for the majority of the cases). Intuitively, one would expect that these variables would be related, in the sense that: (a) the more code is added, the more Technical Debt issues are expected to be introduced; and (b) inexperienced developers would introduce more TD. However, these two factors do not appear to play a key role in the introduction of technical debt. Instead, other confounding factors may be more relevant, such as: (a) the maturity of the project; (b) the developers' habits; or (c) the specific type of tasks performed in each time period. These observations are more evident for the projects that come from the ASF or EF ecosystems, rather than isolated projects. We discuss these directions for further investigation in Section 5.6.2.

5.6.2 Implications to Researchers and Practitioners

Based on the results we are able to provide some first implications to both researchers and practitioners. Regarding **researchers**, we can claim that the accumulation of new code Technical Debt issues reflects (at least to some extent) the characteristics of the development process: by being stable in most cases, the introduction of new code technical debt is probably less related to external factors, and primarily dependent on the capabilities of the team. However, for a non-negligible number of projects, timing seems to be an important factor for studying the accumulation of technical debt: Technical Debt issues do not seem to be

uniformly introduced along evolution, but rather behave as a temporal phenomenon, with multiple and (in some cases) large fluctuations. Therefore, we propose:

- For stable projects, researchers can further investigate the relation between the stable rate of introduction of new code Technical Debt issues with the practices followed by the developers. It would also be valuable to compare stable projects, but with different trends (increasing vs. decreasing), with respect to their key properties.
- For sensitive projects, researchers can perform explanatory studies to unveil the reasons for which spikes occur in the evolution of the introduced technical debt. Such studies could identify possible reasons (e.g., changes in used libraries or frameworks, impact of business goals) that lead teams/projects with a rather stable accumulation of technical debt, to perform worse under certain circumstances. Furthermore, TD fluctuation can be studied in relation to the number of bugs and issues of an evolving software project, investigating whether spikes in TD introduction cause an excessive number of defects. Based on the findings of our study, such questions would be more easily answered in ASF or EF projects, since for their case the correlation of fluctuation and commit density is lower, leaving more space for exploring other parameters.
- Based on the output of the above, researchers can work on more accurate technical debt prevention methodologies that will attack the heart of the problem, based on the particular conditions of each project. For example, a project that is expected to undergo staff turnover, or will face tight deadlines, should calibrate its quality gates to ensure technical debt does not grow beyond thresholds.

Regarding *practitioners*, we encourage them to perform fluctuation analysis and investigate the reasons for the existence of high or frequent peaks in the evolution of introduced Technical Debt issues. Understanding the consequences of their way of working in certain periods (which might lead to excessive accumulation of technical debt) can prove beneficial for process improvement purposes and quality control.

5.7 Threats to Validity

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

Construct Validity reflects how far the examined phenomenon is connected to the intended objectives. The main threat is related to the accuracy by which technical debt can be captured by static analysis tools such as SonarQube. Rule violations reported as Technical Debt issues are only one manifestation of actual code and design inefficiencies. Furthermore,

it is known that such tools are not capable of identifying architectural problems or other types of technical debt such as requirements, test or build debt. In addition, we consider only technical debt that can be mapped to methods, thus ignoring changes which might occur at the level of files. However, while SonarQube is by far not perfect in identifying technical debt, other static analysis tools suffer from similar limitations.

Another construct validity threat is related to the use of the number of commits as a surrogate of the workload that has been carried out by the project participants. Since in open-source projects, voluntary contribution is interleaved with the rest of the developers' activities, we acknowledge that a 'busy' or 'relaxed' period in terms of commits, does not necessarily reflect the actual work conditions of the developers. Moreover, commits differ in the amount of work that they carry: some commits might be accompanied by many changes in several files while other are related to only a few changes. Further research is required to derive the actual workload of developers committing to an open-source software project.

Reliability reflects whether the study has been conducted and reported in a way that others can replicate it and reach the same results. To mitigate this threat, the study protocol is explicitly described listing all data collection and analysis steps. The only subjective data interpretation concerns the identification of spikes (which however is of secondary importance); therefore, to a large extent, researcher bias has been avoided. A replication package³ is available with all available data to allow for an independent replication of the investigation.

External Validity examines the applicability of the findings in other settings, e.g., other software projects, other programming languages and possibly other technical debt tools. We have focused only on Java Apache and Eclipse projects that use Maven as a build tool. This limits the ability to generalize the findings to other projects. The fact that the study focuses on 47 projects of the Apache Software Foundation and Eclipse Foundation, which are highly active and popular among software developers partially mitigates threats to generalization. Nevertheless, replication studies on the effect of new code on the evolution of technical debt are needed to strengthen the validity of the derived conclusions.

5.8 Conclusions and Future Work

Studying the phenomenon of introducing code Technical Debt issues is a research direction that is important for building tools aimed at preventing the accumulation of technical debt. In this study, we focus on code technical debt, and in particular, we explore the temporality of the technical debt introduction phenomenon. To this end, we explore if the introduction of Technical Debt issues changes in different time periods, and if these changes can be attributed to the developers' activity or experience in the corresponding period. To explore these two questions, we have performed a case study on the complete evolution of forty-seven projects from the Apache Software Foundation and Eclipse Foundation.

The results of the study suggested that for the majority of the projects the evolution on

technical debt introduction is stable, i.e., there are not many (at maximum 2) high fluctuations in Technical Debt issues introduction, due to new code. However, a non-negligible part of projects (approx. 40%) present high and frequent fluctuations. These results suggest that technical debt introduction is only partially a temporal phenomenon, with more technical debt being introduced in some time periods. The additional exploration of the phenomenon led to the conclusion that the spikes in the evolution of technical debt introduction are not correlated with spikes in the development activity, nor with the average developers' experience. The findings suggest that the number of commits in a particular period and the maturity of the developers are not the main factors affecting the introduction of 'excessive' technical debt.

The current study focused only on the impact of new code on the accumulation of TD. However, TD can be introduced or removed along method modification as well. Modified methods are much more challenging to analyze as they entail multiple types of changes such as code addition, removal, modification and refactoring. Nevertheless, it would be of great interest to study the impact of new vs. modified code on the system technical debt along software evolution. Furthermore, the study of the relation between introduced TD issues and developer experience can be performed at a finer-grain level by considering the TD issues introduced per commit rather than over an entire time window. Analyzing TD at commit level along with the study of the corresponding commit messages could also shed light into the reasons that lie beneath the excessive introduction of TD issues, with the potential of revealing some of the root causes of TD accumulation.

Table 5.3: SMF of TD issues on New Code

	Project	Mean	Min	Max	Std. Dev.	SMF
Apache Software Foundation	Atlas	475.551	91	1548	407.787	0.538
	Beam	786.081	174	1499	300.519	0.509
	Calcite	470.737	7	2736	692.867	11.902
	Cayenne	94.931	9	353	89.289	1.019
	Commons IO	7.417	0	51	11.638	1.344
	CXF	270.382	0	1008	250.760	0.762
	DeltaSpike	23.936	0	60	16.195	1.396
	Drill	499.488	219	779	158.141	0.335
	Dubbo	98.047	0	447	139.982	1.900
	Flink	973.857	0	3444	725.942	4.080
	Flume	140.952	7	827	208.945	0.340
	Giraph	100.333	0	473	124.362	1.174
	Jackrabbit	351.917	0	2723	611.492	1.639
	jclouds	205.800	6	1265	301.562	0.467
	Knox	123.043	25	375	79.353	0.301
	Kylin	917.702	164	1822	413.281	0.343
	Metron	254.000	71	538	124.613	0.162
	MyFaces	153.248	0	695	183.245	2.992
	NiFi	806.361	0	2985	887.572	0.286
	oozie	171.073	0	1551	380.715	0.451
	OpenWebBeans	71.133	0	760	166.517	0.492
	PDFBox	119.748	0	640	142.110	3.505
	Pulsar	612.500	223	1339	380.892	0.456
	SIS	366.990	0	1340	349.143	9.558
	Storm	598.019	155	1449	323.889	0.389
	TinkerPop	606.025	167	1320	347.057	0.156
	Zeppelin	351.933	0	850	161.070	0.320
Eclipse Foundation	Hono	129.609	47	241	44.706	0.297
	JGit	156.904	0	683	160.817	0.544
	JNoSQL	103.133	0	377	122.967	1.867
	Kapua	322.047	24	1013	296.027	1.972
	Leshan	48.159	3	157	37.433	0.554
	Milo	529.947	1	1479	617.112	44.313
	Tycho	20.610	0	133	35.420	2.441
	Vorto	172.983	14	428	117.235	1.837
Other OSS	apollo	106.393	0	508	145.291	0.700
	gson	5.180	0	26	7.311	0.537
	hutool	861.571	250	1695	540.626	0.020
	jedis	54.864	0	302	77.539	0.795
	litemall	237.387	0	879	246.261	0.646
	mybatis-3	23.071	0	97	27.852	0.696
	seata	568.409	111	1265	437.546	2.273
	spring-boot-admin	25.844	1	117	25.167	5.144
	spring-cloud-alibaba	1211.273	274	2031	626.828	0.378
	webmagic	34.189	0	216	61.224	0.020
	xxl-job	80.228	0	275	54.595	0.506
	zheng	309.737	0	979	458.321	0.353

Table 5.4: Correlation of Commit Activity and TD Introduction

	Project	No. of Commits	Corr. Coef.	Sig. Level
Apache Software Foundation	Atlas	1454	0.62	0.00
	Beam	2780	0.44	0.01
	Calcite	1448	0.53	0.00
	Cayenne	2116	0.45	0.00
	Commons IO	1059	0.50	0.00
	CXF	5078	0.56	0.00
	DeltaSpike	513	0.70	0.00
	Drill	1316	0.65	0.00
	Dubbo	728	0.91	0.00
	Flink	5329	0.44	0.00
	Flume	789	0.70	0.00
	Giraph	668	0.73	0.00
	Jackrabbit	4143	0.82	0.00
	jclouds	4323	0.89	0.00
	Knox	1033	0.42	0.00
	Kylin	3205	0.57	0.00
	Metron	502	0.69	0.00
	MyFaces	1210	0.61	0.00
	NiFi	1490	0.64	0.00
	oozie	587	0.82	0.00
	OpenWebBeans	1583	0.79	0.00
	PDFBox	3758	0.23	0.02
	Pulsar	1503	0.72	0.00
	SIS	828	0.67	0.00
	Storm	738	0.03	0.86
	TinkerPop	5178	0.85	0.00
	Zeppelin	1562	0.13	0.31
Eclipse Foundation	Hono	1977	0.79	0.00
	JGit	1747	0.47	0.00
	JNoSQL	1384	0.89	0.00
	Kapua	1912	0.31	0.04
	Leshan	808	0.52	0.00
	Milo	261	0.80	0.00
	Tycho	290	0.42	0.00
	Vorto	1051	0.33	0.01
Other OSS	apollo	2487	0.75	0.00
	gson	1485	0.57	0.00
	hutool	1937	0.93	0.00
	jedis	1801	0.73	0.00
	litemall	1121	0.83	0.00
	mybatis-3	3726	0.58	0.00
	seata	1230	0.47	0.03
	spring-boot-admin	1540	0.39	0.00
	spring-cloud-alibaba	1748	0.95	0.00
	webmagic	1676	0.93	0.00
	xxl-job	1676	0.20	0.14
	zheng	1241	0.93	0.00

Table 5.5: Correlation of Developers' Experience and TD Introduction

	Project	AVG Dev. Experience	Correl. Coeff.	Sig. Level
Apache Software Foundation	Atlas	825.36	-0.20	0.17
	Calcite	1428.35	0.16	0.16
	Cayenne	1504.98	0.15	0.16
	Commons IO	1391.29	0.01	0.95
	CXF	1280.04	-0.45	0.00
	DeltaSpike	2112.66	-0.58	0.00
	Drill	1134.51	0.06	0.72
	Dubbo	663.85	0.74	0.00
	Flink	1040.61	0.37	0.00
	Flume	1351.80	-0.69	0.00
	Giraph	1230.38	-0.57	0.00
	Jackrabbit	1288.87	-0.52	0.00
	jclouds	1569.23	-0.83	0.00
	Knox	1158.23	-0.33	0.00
	Kylin	1102.61	-0.63	0.00
	Metron	2195.25	-0.92	0.00
	MyFaces	1105.78	-0.16	0.08
	NiFi	1875.56	0.48	0.00
	oozie	845.07	0.69	0.00
	OpenWebBeans	1455.61	-0.28	0.00
	PDFBox	765.44	-0.20	0.04
	Pulsar	2605.61	0.69	0.00
	SIS	1194.91	-0.13	0.22
	Storm	1331.29	-0.22	0.11
	TinkerPop	1833.53	-0.93	0.00
	Zeppelin	1441.31	0.09	0.51
Eclipse Foundation	Hono	1542.89	0.52	0.00
	JGit	1194.92	-0.57	0.00
	JNoSQL	2573.29	-0.31	0.09
	Kapua	1528.83	0.66	0.00
	Leshan	2246.96	-0.45	0.00
	Milo	2395.93	-0.54	0.02
	Tycho	2349.21	-0.38	0.00
	Vorto	799.71	0.38	0.00
Other OSS	apollo	2483.87	-0.77	0.00
	gson	2298.10	-0.03	0.85
	hutool	2889.89	0.97	0.00
	jedis	1185.25	-0.45	0.00
	litemall	2308.54	-0.50	0.00
	mybatis-3	1526.35	-0.43	0.00
	seata	719.72	0.20	0.38
	spring-boot-admin	1093.75	-0.01	0.96
	spring-cloud-alibaba	1915.31	-0.07	0.83
	webmagic	1630.11	-0.24	0.05
	xxl-job	1160.15	-0.27	0.05
	zheng	1371.94	-0.40	0.09

Published as:

G. Digkas, A. Ampatzoglou, A. Chatzigeorgiou, P. Avgeriou, O. Matei, and, R. Heb, "The Risk of Generating Technical Debt Interest: A Case Study," *SN Computer Science*, vol. 2, no. 1, pp. 1-12, 2021, doi: 10.1007/s42979-020-00406-6.

Chapter 6

The Risk of Generating Technical Debt Interest: A Case Study

Abstract

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

6.1 Introduction

In software engineering, the concept of Technical Debt draws an analogy between shortcuts in development and taking out a loan (Cunningham 1992). In particular, the metaphor considers that a software development organization (intentionally or unintentionally) limits the development time/resources through shortcuts, and thus saves a specific amount of money (amount of loan–**TD Principal**) (Alves et al. 2016, Ampatzoglou et al. 2015). This benefit comes with an associated cost, as the product is released with sub-optimal quality, leading to the occurrence of excessive maintenance costs (Harrington 1987); such costs are termed **TD Interest** and include the additional costs for bug fixing, understanding the existing code, adding new features etc. (Alves et al. 2016, Ampatzoglou et al. 2015). While TD Principal is deterministic, TD interest is probabilistic: we are not sure how frequently and to what extent a software artifact will change in the upcoming versions (thus generating interest). The probability of an artifact to generate interest is termed **TD Interest Probability** (Seaman and

Guo 2011). The generation of interest plays a crucial role for the impact of TD on software maintenance. Modules that are rarely maintained, do not cause real problems along software evolution, even if they suffer from high TD; paying back the TD in such cases might be unnecessary. On the contrary, modules with high levels of TD that are often maintained, can cause severe overhead when performing future changes. Such overheads can be so large that in some cases can constitute the maintenance of the software system unsustainable. Based on this, we consider the generation of interest as a *risk* that threatens software maintenance.

In this study we propose a metric, namely **Interest Generation Risk Importance (IGRI)**, to estimate the risk of interest generation. According to Barry Boehm (Boehm 1991), the importance of a risk can be calculated as the product of its impact and likelihood to occur. In the case of IGRI, the likelihood of the risk corresponds to interest probability, whereas the impact to the amount of technical debt interest that it will generate. The proposed metric can be useful in a number of ways; in this study we validate two of them. The first is to assist TD Prioritization, i.e. the priority to refactor a software artifact (Li et al. 2015). Artifacts that pose a more important risk to generate TD interest, would be more urgent for refactoring, so as to prevent excessive maintenance costs. The second is to assess the effect of writing clean new code on the technical debt evolution of the system. If new code is less risky to generate interest, the sustainability of the system can be improved by the addition of clean new code. The clean code paradigm is supported in the literature as an alternative to refactoring for the improvement of software quality (Martin 2009), and it tends to be preferable from the developers' side, as a means of containing technical debt (Arvanitou et al. 2019).

The research outcomes reported in this study has been conducted in the context of the SDK4ED¹ project, funded by the European Union's Horizon 2020 research and innovation programme. The goal of the project is to investigate tradeoffs between optimizations applied to improve Technical Debt, Security and Energy dissipation in software intensive systems. Furthermore, the SDK4ED platform (Jankovic et al. n.d.) aims at assisting decision making with respect to investments on software improvements. The assessment of artifacts which pose a high risk of generating TD interest outlined in this study, is aligned with the overall goal of the project to narrow down the recommended refactoring opportunities. Choosing among optimizations to mitigate software vulnerabilities detected through static analysis (Siavvas et al. 2018), to improve performance (Siavvas and Gelenbe 2019, Siavvas and Gelenbe 2019, Gelenbe and Zhang 2019) and energy consumption (Tsimpourlas et al. 2018, Marantos et al. 2018, Xygkis et al. 2018), to improve software maintainability (Charalampidou et al. 2018, Skiada et al. 2018, Kouros et al. 2019, Nikolaidis et al. 2019, Digkas, Chatzigeorgiou, Ampatzoglou and Avgeriou 2020), and to manage architectural technical debt (Sas et al. 2019) is a non-trivial task. Research has proved the existence of interrelations between these qualities (Papadopoulos et al. 2018, Siavvas, Marantos, Papadopoulos, Kehagias, Soudris and Tzovaras 2019, Siavvas, Tsoukalas, Jankovic, Keha-

¹<https://sdk4ed.eu/>

gias, Chatzigeorgiou, Tzovaras, Anicic and Gelenbe 2019) rendering the extraction of the best possible sequence of software refactoring subject to a Multi-Criteria Decision Making (MCDM) analysis which has been implemented in the SDK4ED platform, as well as an approach for treating refactorings as a financial investment (Ampatzoglou et al. 2020).

We organize the rest of the paper as follows. In Section 6.2, we present: (a) related work on technical debt prioritization; and (b) background work on software risk management. The framework that we use for calculating Technical Debt Interest and Interest Probability, as well as the proposed metric for assessing the importance of interest generation as a risk are introduced in Section 6.3. In Section 6.4, we present the empirical design through which we explore the two aforementioned usage scenarios of the IGRI metric. In Section 6.5 we answer the research questions, which we discuss in Section 6.6, by interpreting the results and providing implications for researchers and practitioners. Finally, in Section 6.7 we present the threats to validity of this study accompanied by mitigation actions, whereas in Section 6.8 we conclude the paper.

6.2 Related Work and Background Information

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

TD Prioritization is the technical debt management activity that aims at ranking the TD items that have been identified, based on their urgency to be resolved (Li et al. 2015). According to Li et al. (Li et al. 2015), TD Prioritization has been studied in 18% of the TD research corpus. TD Prioritization methods can be discussed under two perspectives: based on the concepts used as inputs, as well as, based on the approach used for prioritization per se. With respect to inputs (Seaman and Guo 2011), TD prioritization can be performed based on the main TD metrics (i.e., TD principal, interest, or interest probability). With respect to approach, existing methods for TD prioritization can be categorized into four main classes: (a) cost-benefit-based approaches; (b) principal-based approaches; (c) portfolio management approaches; and (d) interest-based approaches.

Software Risk Management is the process that aims at: (a) identifying the risks in software development; (b) determining their importance; and (c) suggesting mitigation actions (Boehm 1989). According to Boehm, there are three main categories of risks: project risks, product risks, and business risks (Boehm 1991). Among these categories, the generation of Technical Debt Interest falls in the product risk category: i.e., “*risks that affect the quality or performance of the software being developed*”. In this paper, we focus on **Risk Analysis** (see Sommerville (Sommerville 2011)): risk analysis aims at assessing the likelihood and consequences of all risks identified in a system. Therefore, the rest of this sub-section focuses on

how risks can be assessed. In the literature, there are two main schools for risk assessment: categorical risk assessment and continuous risk assessment. Regarding categorical risk assessment, according to Sommerville (Sommerville 2011), risk analysis relies on judgement and experience to find the probability of a risk (rare, unlikely, possible, likely, or almost certain) and the effects of the risk (catastrophic, major, moderate, minor, or negligible). Based on this the project managers generate a table according to seriousness of risk and update it during each iteration of the risk process, as shown in Figure 6.1.

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

Figure 6.1: Risk Assessment Matrix

With respect to continuous risk assessment, in a seminal work of software project management, Boehm (Boehm 1991) introduced the basic principles of software risk management. As part of risk analysis, he suggests that risk exposure (also termed as risk impact) can be calculated as the product of the probability of unsatisfactory outcome and the loss caused by the unsatisfactory outcome.

6.3 Assessing the Risk of Generating Interest

This section focuses on tailoring the software risk management concepts to fit the technical debt metaphor. IGRI is meant to quantify the impact of a possible change in a specific artifact

that suffers from technical debt, by assessing the probability of this artifact to change and the amount of interest that is going to be generated, upon such a tentative change. The rationale of the metric relies on the risk importance formula, as proposed by Boehm (Boehm and Sullivan 2000). The rest of this section describes the way that TD Interest and TD Interest Probability are calculated.

$$IGRI = Interest \times InterestProbability \quad (6.1)$$

6.3.1 Technical Debt Interest

The estimation of TD Interest is a challenging endeavor as interest refers to the “additional” maintenance costs, incurred by inefficiencies in software; whereas the nominal software maintenance effort (i.e. the effort that would have been required in case the system had zero technical debt) is unknown. The calculation of Technical Debt Interest in this study relies on the FITTED framework, which has been proposed (Ampatzoglou et al. 2015) and empirically validated in our previous work (Ampatzoglou et al. 2018, Tsintzira et al. 2019). FITTED attempts to calculate interest by estimating the level of “sub-optimality” of any given software artifact. In particular, FITTED assumes that the effort required to maintain a sub-optimal artifact is analogous to the ratio of the maintainability of the two artifacts:

$$Effort_{opt} = Effort_{act} \times \frac{maintainability_{act}}{maintainability_{opt}} \quad (6.2)$$

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

$$TDInterest = Effort_{act} \times (1 - \frac{maintainability_{act}}{maintainability_{opt}}) \quad (6.3)$$

Maintainability. To calculate the maintainability of a software artifact, we first identify a set of other artifacts (Ampatzoglou et al. 2018)) that can be considered (structurally) similar. Artifact similarity is assessed based on their lines of code, number of methods, cognitive complexity. Second, we calculate the optimal value of selected metrics among the set of similar artifacts. The best metric scores (highest or lowest values, based on the type of the metric) are assumed to characterize the hypothetical ‘*optimal*’ which the artifact under study could potentially reach. A simplified example is outlined in Fig. 6.2.

Finally, we assess maintainability as the average ratio of each metric score of the artifact under study, against the optimal value. As maintainability metrics, we have used ten well-established maintainability predictors (Depth of Inheritance Tree–DIT, Number of Children

Figure 6.2: Notion of hypothetical optimal among similar artifacts

Classes–NOCC, Message Passing Coupling–MPC, Response for a Class–RFC, Data Abstraction Coupling–DAC, Lack of Cohesion of Methods–LCOM, Cyclomatic Complexity–CC, Weighted Method per Class–WMPC, Lines of Code–LOC, and Number of Properties–SIZE2), from the metric suites of Chidamber and Kemerer (Chidamber et al. 1998), and Li and Henry (Li and Henry 1993). The aforementioned list of metrics has been acknowledged by Riaz et al. (Riaz et al. 2009) as the most prominent maintainability predictors.

Maintenance Effort. Predicting how a system will evolve is a non-trivial task, in the sense that none can foresee with certainty the changes that will be requested by customers, bug fixing activities, etc. Hence, we follow a simple, yet relatively reasonable approach and assess future maintenance effort, based on historical change data. In particular, for each artifact we calculate the average number of code lines which have been introduced, removed or changed, for every transition from one revision to the next, and use this as a proxy of maintenance effort in each future commit. Finally, we project this average maintenance effort to future releases of the analyzed artifact. A similar approach was employed in various other research works (Conejero et al. 2018, Kazman et al. 2015).

6.3.2 Technical Debt Interest Probability

Interest probability is calculated based on past maintenance data. To this end we use the Percentage of Commits in which a Class has Changed (PCCC) metric (Arvanitou et al. 2017).

Despite the fact that the Effort of Maintenance and PCCC are related, the two measures correspond to different views of the same phenomenon, in the sense that Maintenance Effort captures lines of code (i.e. the average extent of change), whereas PCCC a number of commits (frequency of changes). Thus, we consider them independent and suitable for being used in the same calculation.

6.4 Case Study Design

The case study is designed and reported based on the linear-analytic structure as described by Runeson et al. (Runeson et al. 2012). This section presents the study design in detail.

6.4.1 Research Goals and Questions

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

RQ1: Is IGRI able to prioritize TD items similarly to an expert? To answer this research question, we calculate IGRI for all classes of a project and we record the urgency to fix TD (specifically to improve the quality of individual classes), based on the expert opinion of software engineers. A correlation analysis between the IGRI values and the expert opinions could validate or invalidate IGRI as a suitable prioritization indicator. In case IGRI is able to resemble the expert opinion with a strong correlation, we would be able to resolve an important scalability problem in TDM, since experts cannot afford to assess hundreds or even thousands of artifacts manually. Using IGRI, they would instead get automated suggestions on which TD items to check first for refactoring opportunities.

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

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

component to produce interest. Subsequently, one could advocate the writing of clean new code as a way to decrease the risk of generating interest, effectively reversing the negative effect of TD.

6.4.2 Cases and Units of Analysis

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

6.4.3 Data Collection

To address the research questions the following process has been followed. In the **first step**, we initially analyzed the MaQuali source code with the SDK4ED toolkit² and **quantified IGRI (Interest Generation Risk Importance)** for every class of the software. Then, we aggregated the results at the level of packages. Next, we randomly picked³ 10 packages and asked Holisun’s engineers to provide a ranking of these packages in terms of maintenance risk. This process has led us to the dataset outlined in Table 6.1. The first column corresponds to the name of the package, the second column to interest (per commit), the third to interest probability, and the fourth to IGRI.

Table 6.1: TD Interest Assessment of MaQuali Packages

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

As a **second step**, we **asked the software engineers** of Holisun that focus on MaQuali maintenance to rank the aforementioned packages, based on the following question: “*Please rank the aforementioned packages (ties are acceptable—however, not preferable) in terms of the risk that their maintenance might lead to extreme delays. As maintenance, please*

²<https://sdk4ed.eu>

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

consider the time that you spend for adding a new requirement, for fixing a bug, etc. In this question, consider not only the time required for one maintenance action, but also how frequently you need to maintain them. Assign 1 to the package that is the least risky and 20 to the most risky packages". Packages have been shuffled for each respondent, while the assessments of each package, based on the SDK4ED platform was hidden from the engineers. The analysis of the respondents' answers (5 software engineers) have been aggregated.

As a **third step**, we have performed the **analysis of new code technical debt**, similarly to our earlier work (Digkas, Ampatzoglou, Chatzigeorgiou and Avgeriou 2020). For a software system evolving through a number of revisions we track new methods introduced either in entirely new packages or in existing classes. We then compute the quality of these new methods in terms of their technical debt by linking identified technical debt issues to these methods. Note that we use the concept of $TD_{density}$, which is the technical debt of these methods normalized over their size in lines of code. $TD_{density}$ enables the comparison of technical debt between artifacts of different sizes (such as new methods vs. the already existing system).

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

Step 1: Commit Retrieval

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

Step 2: Linking technical debt issues with methods

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

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

Step 3: Tracking new methods

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

1. *New Methods in New Files:* For files which are new in each revision, we tag all of their methods as new. The methods of each file are obtained with the use of their Abstract Syntax Tree.
2. *New Methods in Modified Files:* For files which have been modified in a revision, we identify entirely new methods using the Gumtree Spoon AST Diff tool⁴.

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

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

Impact of new methods

$$\Delta TD_{density}(new) = \frac{TD_{t-1} + TD_{new(t)}}{LOC_{t-1} + LOC_{new(t)}} - TD_{density}(t-1) \quad (6.4)$$

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

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

6.4.4 Analysis

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

6.5 Results

In this section, we present the results of the case study, organized by research question. In particular, in Section 6.5.1, we present the results on the ability of IGRI to predict the risk

⁴<https://github.com/SpoonLabs/gumtree-spoon-ast-diff>

of software packages to produce high interest. Subsequently, in Section 6.5.2, we use the newly proposed index to assess the impact of new code on the risk of producing technical debt interest.

6.5.1 Ability of IGRI to Prioritize TD Artifacts

As a means for validating the ability of IGRI to estimate of the risk of a package to generate Technical Debt Interest, we contrast the metric to the perception of stakeholders on the risk of package maintenance to lead to extreme risks. To achieve this goal, since: (a) the two variables have a different value range; and (b) we focus on prioritization instead of prediction, we preferred to treat the two variables as ordinal ones. Thus, we transformed them to the rank that corresponds to a specific value (i.e., the highest IGRI is assigned the value 1; whereas the lowest IGRI is assigned the value 10). To visualize the ability of IGRI to consistently rank packages, based on their risk to produce interest (**metric property**: consistency (Committee et al. 1998)), in Figure 6.3, we present a scatter plot.

As it can be observed from Figure 6.3, the ranking is almost consistent (the dots are close to the main diagonal line), with only some exceptions. The majority of deviations is by one rank, with only one exception (**package**: `fr.icms.printing`). The specific package is ranked as *high risk* according to stakeholders, but as *low risk*, based on IGRI. According to a lead developer of the MaQuali software: “*the printing package was difficult to maintain and keep sustainable, because of the lack of strong printing support for java programs (especially custom HTML printing)*”. The fact that the interest probability of this package is quite low, we can infer that the opinion of stakeholders on risk is more related to Technical Debt Interest (i.e., maintenance difficulty) rather than maintenance frequency.

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

To explore if the aforementioned results are statistically significant, we performed a Spearman Rank correlation analysis. The results (**correlation coefficient**: 0.827 and **sig**: 0.003) of the test suggest that the two ranks are very strongly correlated (and statistically significant). Thus, an IGRI-based prioritization can safely subsume (Committee et al. 1998) the ranking that an experienced practitioner would provide to packages in terms of risk to

Figure 6.3: Risk Ranking Consistency of IGRI and Perception of Stakeholders

generate Technical Debt Interest. This outcome is significant, in the sense that IGRI calculation is automated; therefore, it can easily scale at large codebases, and is unbiased from stakeholders experience. Thus, inexperienced developers can use the ranking and identify maintainability challenges similarly to more experienced developers.

6.5.2 Relation of IGRI and new code

In this section, we use the IGRI metric, so as to explore the relation between new code and the risk of generating TD Interest. New code has been discussed in the literature as an alternative to refactoring, for reducing the amount of technical debt (Digkas et al. 2017, Zabardast et al. 2020). To this end, we explore: (a) if the average percentage of new code that is

accumulated in a package along evolution, is associated with a decrease or increase of IGRI–RQ2.1; and (b) if there is a relation between IGRI and the quality of the new code–RQ2.2.

Regarding RQ2.1, we first explore if the average percentage of new code (in all versions) against all lines of code of the package, is correlated to IGRI of the package (calculated as the average value of the IGRI score of its classes). The results suggest that there exist a very strong (**correlation coefficient**: -0.745 and **sig**: 0.012) negative relation, that is statistically significant. The negative sign of the relation suggests that the less new code is introduced in a package, the higher the risk of the corresponding package generating interest. To visualize the aforementioned relationship, in Figure 6.4, we present the boxplots of IGRI for each percentile (quartile) of new code density. For instance the first percentile corresponds to packages that 0% - 25% of their code in each version (on average) is new. Based on Figure 6.4, the median IGRI for the packages of this group is approx. 8 (**mean value**: 13.84); whereas the median IGRI for packages in which new code accounts for 26% - 50% of their codebase the median is almost zero (**mean value**: 1.09).

Figure 6.4: Percentage of New Code and Risk of Producing Interest

Regarding RQ2.2, rather than focusing on the amount of new code that is added, we focus on the quality of the new code. Thus, we correlated the $TD_{density}$ of the new code with IGRI. The results of the Spearman correlation, suggest a moderate negative correlation (**correlation coefficient**: -0.547 and **sig**: 0.100); which, however, is not statistically significant. This result, also suggests that new code of better quality tends to reduce the risk of generating interest, but this result cannot be generalized. To visualize the difference, we split the dataset into two groups (poor quality- $TD_{density} > 1.0$ and good quality- $TD_{density} < 1.0$) and provide the boxplots of IGRI—see Figure 6.5. Despite the difference in the mean values (2.55 for Good Quality Code vs. 7.49 for Poor Quality Code), the two samples do not differ statistically significantly. However, this could be due to the small sample size of our case study.

Figure 6.5: Quality of New Code and Risk of Producing Interest

By synthesizing the results of RQ2.1 and RQ2.2 we can claim that new code is associated with the risk of generating interest: the more new code is inserted along evolution the lower the risk, and if this new code is "clean" the impact of the Technical Debt Interest Risk is

further reduced. This result, complies with the literature, suggesting that clean new code reduces the amount of Technical Debt Principal along evolution (Digkas et al. 2017, Zabardast et al. 2020). Additionally, we emphasize that the amount of new code appears to be a more important factor for reducing the risk of producing Technical Debt Interest, compared to the quality of the code. This finding is surprising as code of better quality would be expected to decrease the risk of heavy maintenance; thus it deserves further investigation.

6.6 Discussion

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

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

6.7 Threats to Validity

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

Construct Validity refers to the extent by which the study succeeds in properly linking

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

Reliability, which is related to the validity of conclusions, assesses whether the findings might be heavily dependent on the involved researchers and their methods. To mitigate any reliability threats we report all steps followed to obtain the dataset for the investigated research questions and provide links to the employed tools. Moreover, the employed dataset along with the variable values for the statistical analysis are available in a replication package⁵.

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

6.8 Conclusions

Acknowledging that efficient prioritization of technical debt repayment is key for software sustainability, we have introduced a simple, yet effective way to estimate risk importance of technical debt items. By considering both the amount of technical debt interest as well as the probability of artifacts to undergo changes, we have proposed the Interest Generation Risk Importance (IGRI) measure. IGRI quantifies the impact of a possible change in a specific artifact that suffers from technical debt.

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

⁵ <https://drive.google.com/drive/folders/1c2RX6KmmBCLoU-ac2uEPxc5F2NMISgix>

Chapter 7

Conclusions and Future Work

This chapter discusses the main conclusions reached through this PhD project and some directions to further this work. In particular in Chapter 7.1, we answer the research questions, and highlight the main contributions. In Chapter 7.2, we consider future work.

7.1 Answers to Research Questions

The problem statement, in which this thesis relies upon, as stated in Chapter 1.3.1, is the following: “*The code Technical Debt (TD) that exists in a software system is detrimental to the maintenance and evolution of that system, to the extent that the system may become unsustainable. Software engineers can either repay the TD through refactoring the code, or they can prevent the TD from occurring in the first place, through writing clean new code. Which of the two ways is followed in practice and how?*”. To address this problem, we have asked five main research questions, which were subsequently answered in Chapters 2 to 6.

RQ1: How does code Technical Debt evolve over time?

To answer this research question we have performed a large-scale case study on 16000 commits of 66 open-source software projects. Based on the data, we have learned that in the majority of the systems that we studied, there is a significant increase trend on the size, number of issues, and on the complexity metrics of the project. On the other hand, the normalized code technical debt ($TD_{density}$) decreases as the project evolves. The improvement of $TD_{density}$ is an important finding, in the sense that it contradicts the 7th of Lehman’s laws of evolution which states that the quality of software deteriorates over time. This finding gave rise to further examining what practices can lead to a decrease in the $TD_{density}$ of a software system: through repayment (refactoring) or preventing (clean new code).

RQ2: Is code Technical Debt repayment an established approach in open-source software development?

To answer this question we have explored a data-set obtained by analysing 57 open-source software projects, focusing on the amount of code technical debt that is repaid through refactoring (issues that are fixed). The results revealed that: (a) a large percentage of code technical debt is paid back during the evolution; however, for the majority of the projects the fixing rate is below 30%, and (b) a small percentage of code Technical Debt Issue types (such as complex methods, duplicates and exception handling problems) account for the majority of

code Technical Debt Issue fixes and the majority of code TD that is paid back. This finding confirmed existing literature suggesting that refactoring is not systematically and extensively applied in practice; thus it does not appear to be an effective way to reduce $TD_{density}$.

RQ3: Can practitioners control code Technical Debt Principal along evolution, by writing clean new code?

To answer this research question we have explored the evolution of 24 open-source software projects, characterized each commit based on the type of change that introduced it (new code, modification, or deletion) and calculated the contribution of each change to $TD_{density}$. The whole process was fully automated, and was added in the form of a quality gate in the SDK4ED platform. The results of the empirical study suggested that the quality of new code in terms of its $TD_{density}$, is, for the majority of the revisions, higher than the quality of the system in which the code is introduced. Moreover, among new, deleted, and modified code, the contribution of code modification exhibits a strong association to the change in the system's $TD_{density}$, followed by the contribution of new code. The contribution of new code is more profound for the transitions in which the quality of the system improved. More specifically, it was found that adding new code that is cleaner than the existing code-base coincides very frequently with a reduction in $TD_{density}$ of the system. The same association, has also been observed for the contribution of method modification to the change of the total $TD_{density}$. The findings of this study suggest that writing code that has fewer Technical Debt Issues than the host code, can prove a very efficient and low-cost approach for managing TD. Applying Quality Gates to ensure that each commit yields fewer violations than the average, essentially leads to an improving quality trend, thereby reversing the software ageing phenomenon.

RQ4: Is the amount of code Technical Debt Principal introduced by new code a temporal phenomenon?

To answer this research question we have explored the full history of 29 open-source software projects, by considering 6-month sliding window periods. The results of the study suggested that the introduction of code Technical Debt issues due to the insertion of new code is independent of time, for more than half of the projects. This can be interpreted as an indication of project maturity, in the sense that consistent quality is achieved throughout evolution. However, even for these stable projects, the absence of fluctuations does not necessarily imply the absence of any trend. In contrast, for a subset of the analyzed projects, the introduction of new code Technical Debt issues is a temporary phenomenon, since many spikes exist in their evolution. Finally, the number of commits made in a time period as well as the experience of developers, are not correlated to the number of introduced Technical Debt issues into the system (for the majority of the cases). Intuitively, one would expect that these variables would be related, in the sense that: (a) the more code is added, the more

code Technical Debt issues are expected to be introduced; and (b) inexperienced developers would introduce more code technical debt. However, these two factors do not appear to play a key role in the introduction of code technical debt. Instead, other confounding factors may be more relevant, such as: (a) the maturity of the project; (b) the developers' habits; or (c) the specific type of tasks performed in each time period.

RQ5: Can practitioners control code Technical Debt Interest along evolution, by writing clean new code?

As a final question, we focused on the second pillar of the code technical debt metaphor, namely: TD interest. To answer this question we relied on a industrial project with more than 100K lines of code. To assess interest we introduced a new metric for quantifying the risk of interest, namely the Interest Generation Risk Importance (IGRI). As a first step we have successfully validated IGRI in the industrial setting (correlated more than 82% to experts opinion), and then we assessed the relation between new code and IGRI. The results of this empirical evaluation pointed out that, the more new code is added to a software system, the lower the risk to generate interest, compared to already existing code.

7.2 Ongoing and Future Work

As future work opportunities that have emerged from the research conducted in the context of this PhD thesis, we have identified three main directions: a) Comparison between writing clean new code and refactoring as means for containing code technical debt; b) Personalized assessment of practices leading to code TD introduction; c) Investigating the industrial applicability of the 'clean new code' approach for various types of software systems.

7.2.1 Clean New Code vs Refactorings

In this thesis we have demonstrated the potential of writing clean new code as a means of reducing code TD principal and interest over time. However, the most prominent approach for improving the internal quality of an existing system is the application of refactorings. Considering the empirical evidence that systematic refactoring is rare in industry and open-source software projects we plan to contrast the impact of clean (or cleaner) code commits to that of applied refactorings. To this end we have already started the collection of commit files and the classification of changes to new code introduction (based on the techniques outlined in this thesis) and refactoring applications employing state-of-the-art refactoring miners. Assessing the average impact of cleaner code during method addition and that of refactoring existing methods or classes can shed light into which approach yields the higher benefit both cumulatively and per commit. Such information can potentially reshape development teams' strategies for improving software quality.

7.2.2 Personalized assessment of code TD introduction

The ability to discriminate between types of changes at a fine-grained level and estimate the incurred or averted TD enables the application of similar analysis approaches for the code associated with each developer. By establishing which code or design rule violations are more frequent for each member of the development team allows the study of the phenomenon with a focus on each developers' habits. By letting a Quality Gate differentiate the types of problems introduced by each programmer, personalized proactive recommendations can be made. Similarly, the adoption of gamification approaches (such as badges awarded to developers introducing cleaner new code) can cultivate a healthy software mindset limiting the need for third-party audits and code inspections. We plan to extend the existing tool-set so as to report code TD principal and interest mapped to each developer's activity.

7.2.3 Investigating the industrial applicability of the 'clean new code' paradigm

The testbench for studying the promise of the proposed approaches in this thesis has been primarily the domain of open-source software projects. It goes without saying that the effectiveness of the discussed strategies and especially the potential of writing clean new code to mitigate code TD should be validated in an industrial setting as well. Moreover, with the advent of service-oriented and cloud-based software systems, the applicability of Quality Gates should be studied in the context of individual service creation as well as workflow composition for the cloud. Leveraging the tools that have been developed during the thesis we plan to study the application of a New Code TD analysis process on industrial projects including cloud applications.

Bibliography

- AlOmar, E. A., Peruma, A., Newman, C. D., Mkaouer, M. W. and Ouni, A.: 2020, On the relationship between developer experience and refactoring: An exploratory study and preliminary results, *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pp. 342–349.
- Alshayeb, M.: 2009, Empirical investigation of refactoring effect on software quality, *Information and software technology* **51**(9), 1319–1326.
- Alves, N. S., Mendes, T. S., de Mendonça, M. G., Spínola, R. O., Shull, F. and Seaman, C.: 2016, Identification and management of technical debt: A systematic mapping study, *Information and Software Technology* **70**, 100–121.
- Alves, N. S., Ribeiro, L. F., Caires, V., Mendes, T. S. and Spínola, R. O.: 2014, Towards an ontology of terms on technical debt, *2014 Sixth International Workshop on Managing Technical Debt*, IEEE, pp. 1–7.
- Alves, V., Niu, N., Alves, C. and Valença, G.: 2010, Requirements engineering for software product lines: A systematic literature review, *Information and Software Technology* **52**(8), 806–820.
- Amanatidis, T., Mittas, N., Moschou, A., Chatzigeorgiou, A., Ampatzoglou, A. and Angelis, L.: 2020, Evaluating the agreement among technical debt measurement tools: building an empirical benchmark of technical debt liabilities, *Empirical Software Engineering* **25**(5), 4161–4204.
- Ampatzoglou, A., Ampatzoglou, A., Chatzigeorgiou, A. and Avgeriou, P.: 2015, The financial aspect of managing technical debt: A systematic literature review, *Information and Software Technology* **64**, 52–73.
- Ampatzoglou, A., Ampatzoglou, A., Chatzigeorgiou, A., Avgeriou, P., Abrahamsson, P., Martini, A., Zdun, U. and Systa, K.: 2016, The perception of technical debt in the embedded systems domain: an industrial case study, *2016 IEEE 8th International Workshop on Managing Technical Debt (MTD)*, IEEE, pp. 9–16.

- Ampatzoglou, A., Arvanitou, E.-M., Ampatzoglou, A., Avgeriou, P., Tsintzira, A.-A. and Chatzigeorgiou, A.: 2020, Architectural decision-making as a financial investment: An industrial case study, *Information and Software Technology* p. 106412.
- Ampatzoglou, A., Michailidis, A., Sarikyriakidis, C., Ampatzoglou, A., Chatzigeorgiou, A. and Avgeriou, P.: 2018, A framework for managing interest in technical debt: an industrial validation, *Proceedings of the 2018 International Conference on Technical Debt*, pp. 115–124.
- Arcoverde, R., Garcia, A. and Figueiredo, E.: 2011, Understanding the longevity of code smells: preliminary results of an explanatory survey, *Proceedings of the 4th Workshop on Refactoring Tools*, pp. 33–36.
- Arvanitou, E.-M., Ampatzoglou, A., Bibi, S., Chatzigeorgiou, A. and Stamelos, I.: 2019, Monitoring technical debt in an industrial setting, *Proceedings of the Evaluation and Assessment on Software Engineering*, pp. 123–132.
- Arvanitou, E.-M., Ampatzoglou, A., Chatzigeorgiou, A. and Avgeriou, P.: 2016, Software metrics fluctuation: a property for assisting the metric selection process, *Information and Software Technology* **72**, 110–124.
- Arvanitou, E.-M., Ampatzoglou, A., Chatzigeorgiou, A. and Avgeriou, P.: 2017, A method for assessing class change proneness, *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, pp. 186–195.
- Avgeriou, P. C., Taibi, D., Ampatzoglou, A., Fontana, F. A., Besker, T., Chatzigeorgiou, A., Lenarduzzi, V., Martini, A., Moschou, N., Pigazzini, I. et al.: 2020, An overview and comparison of technical debt measurement tools, *IEEE Software* .
- Avgeriou, P., Kruchten, P., Ozkaya, I. and Seaman, C.: 2016, Managing technical debt in software engineering (dagstuhl seminar 16162), *Dagstuhl Reports*, Vol. 6, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
- Bavota, G., Canfora, G., Di Penta, M., Oliveto, R. and Panichella, S.: 2013, The evolution of project inter-dependencies in a software ecosystem: The case of apache, *2013 IEEE international conference on software maintenance*, IEEE, pp. 280–289.
- Beyer, S., Macho, C., Di Penta, M. and Pinzger, M.: 2017, Analyzing the relationships between android api classes and their references on stack overflow, *Technical Report* .
- Boehm, B.: 1989, Software risk management, *European Software Engineering Conference*, Springer, pp. 1–19.
- Boehm, B. and Sullivan, K.: 2000, Software economics: A roadmap, the future of software engineering, *Proceedings of the 22nd International Conference on Software Engineering*, pp. 319–343.

- Boehm, B. W.: 1991, Software risk management: principles and practices, *IEEE software* **8**(1), 32–41.
- Campbell, G. and Papapetrou, P. P.: 2013, *SonarQube in action*, Manning Publications Co.
- Charalampidou, S., Arvanitou, E.-M., Ampatzoglou, A., Avgeriou, P., Chatzigeorgiou, A. and Stamelos, I.: 2018, Structural quality metrics as indicators of the long method bad smell: An empirical study, *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, IEEE, pp. 234–238.
- Chatzigeorgiou, A. and Manakos, A.: 2014, Investigating the evolution of code smells in object-oriented systems, *Innovations in Systems and Software Engineering* **10**(1), 3–18.
- Chávez, A., Ferreira, I., Fernandes, E., Cedrim, D. and Garcia, A.: 2017, How does refactoring affect internal quality attributes? a multi-project study, *Proceedings of the 31st Brazilian Symposium on Software Engineering*, pp. 74–83.
- Chidamber, S. R., Darcy, D. P. and Kemerer, C. F.: 1998, Managerial use of metrics for object-oriented software: An exploratory analysis, *IEEE Transactions on software Engineering* **24**(8), 629–639.
- Committee, S. . S. E. S. et al.: 1998, 1061-1998-ieee standard for a software quality metrics methodology.
- Conejero, J. M., Rodríguez-Echeverría, R., Hernández, J., Clemente, P. J., Ortiz-Caraballo, C., Jurado, E. and Sánchez-Figueroa, F.: 2018, Early evaluation of technical debt impact on maintainability, *Journal of Systems and Software* **142**, 92–114.
- Cunningham, W.: 1992, The wycash portfolio management system, *ACM SIGPLAN OOPS Messenger* **4**(2), 29–30.
- Curtis, B., Sappidi, J. and Szyrkarski, A.: 2012, Estimating the size, cost, and types of technical debt, *2012 Third International Workshop on Managing Technical Debt (MTD)*, IEEE, pp. 49–53.
- Dagenais, B. and Robillard, M. P.: 2011, Recommending adaptive changes for framework evolution, *ACM Transactions on Software Engineering and Methodology (TOSEM)* **20**(4), 1–35.
- Decan, A., Mens, T., Claes, M. and Grosjean, P.: 2016, When github meets cran: An analysis of inter-repository package dependency problems, *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1, IEEE, pp. 493–504.

- Dieste, O., Aranda, A. M., Uyaguari, F., Turhan, B., Tosun, A., Fucci, D., Oivo, M. and Juristo, N.: 2017, Empirical evaluation of the effects of experience on code quality and programmer productivity: an exploratory study, *Empirical Software Engineering* **22**(5), 2457–2542.
- Digkas, G., Ampatzoglou, A., Chatzigeorgiou, A. and Avgeriou, P.: 2020, On the temporality of introducing code technical debt, *International Conference on the Quality of Information and Communications Technology*, Springer, pp. 68–82.
- Digkas, G., Ampatzoglou, A., Chatzigeorgiou, A., Avgeriou, P., Matei, O. and Heb, R.: 2021, The risk of generating technical debt interest: a case study, *SN Computer Science* **2**(1), 1–12.
- Digkas, G., Chatzigeorgiou, A. N., Ampatzoglou, A. and Avgeriou, P. C.: 2020, Can clean new code reduce technical debt density, *IEEE Transactions on Software Engineering* .
- Digkas, G., Lungu, M., Avgeriou, P., Chatzigeorgiou, A. and Ampatzoglou, A.: 2018, How do developers fix issues and pay back technical debt in the apache ecosystem?, *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, pp. 153–163.
- Digkas, G., Lungu, M., Chatzigeorgiou, A. and Avgeriou, P.: 2017, The evolution of technical debt in the apache ecosystem, *European Conference on Software Architecture*, Springer, pp. 51–66.
- Du Bois, B. and Mens, T.: 2003, Describing the impact of refactoring on internal program quality, *International Workshop on Evolution of Large-scale Industrial Software Applications*, pp. 37–48.
- Dueñas, J. C., Cuadrado, F., Santillán, M., Ruiz, J. L. et al.: 2007, Apache and eclipse: Comparing open source project incubators, *IEEE software* **24**(6), 90–98.
- Falessi, D., Russo, B. and Mullen, K.: 2017, What if i had no smells?, *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, IEEE, pp. 78–84.
- Falleri, J.-R., Morandat, F., Blanc, X., Martinez, M. and Monperrus, M.: 2014, Fine-grained and accurate source code differencing, *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pp. 313–324.
- Field, A.: 2013, *Discovering Statistics Using IBM SPSS Statistics*, Sage Publications Ltd.
- Fowler, M.: 2018, *Refactoring: improving the design of existing code*, Addison-Wesley Professional.

- Gelenbe, E. and Zhang, Y.: 2019, Performance optimization with energy packets, *IEEE Systems Journal* **13**(4), 3770–3780.
- Harrington: 1987, *Poor-quality cost: implementing, understanding, and using the cost of poor quality*, CRC Press LLC.
- Hassan, A. E.: 2009, Predicting faults using the complexity of code changes, *2009 IEEE 31st international conference on software engineering*, IEEE, pp. 78–88.
- Herraiz, I., Robles, G., González-Barahona, J. M., Capiluppi, A. and Ramil, J. F.: 2006, Comparison between slocs and number of files as size metrics for software evolution analysis, *Conference on Software Maintenance and Reengineering (CSMR'06)*, IEEE, pp. 8–pp.
- Jaafar, F., Guéhéneuc, Y.-G., Hamel, S., Khomh, F. and Zulkernine, M.: 2016, Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults, *Empirical Software Engineering* **21**, 896–931.
- Jankovic, M., Kehagias, D., Siavvas, M., Tsoukalas, D. and Chatzigeorgiou, A.: n.d., The sdk4ed approach to software quality optimization and interplay calculation.
- Janus, A., Dumke, R., Schmietendorf, A. and Jäger, J.: 2012, The 3c approach for agile quality assurance, *2012 3rd International Workshop on Emerging Trends in Software Metrics (WETSoM)*, IEEE, pp. 9–13.
- Kazman, R., Cai, Y., Mo, R., Feng, Q., Xiao, L., Haziye, S., Fedak, V. and Shapochka, A.: 2015, A case study in locating the architectural roots of technical debt, *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2, IEEE, pp. 179–188.
- Kosti, M. V., Georgiadis, K., Adamos, D. A., Laskaris, N., Spinellis, D. and Angelis, L.: 2018, Towards an affordable brain computer interface for the assessment of programmers' mental workload, *International Journal of Human-Computer Studies* **115**, 52–66.
- Kouros, P., Chaikalis, T., Arvanitou, E.-M., Chatzigeorgiou, A., Ampatzoglou, A. and Amanatidis, T.: 2019, Jcaliper: search-based technical debt management, *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pp. 1721–1730.
- Kruchten, P., Nord, R. L. and Ozkaya, I.: 2012, Technical debt: From metaphor to theory and practice, *Ieee software* **29**(6), 18–21.
- Lehman, M. M.: 1996, Laws of software evolution revisited, *European Workshop on Software Process Technology*, Springer, pp. 108–124.

- Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E. and Turski, W. M.: 1997, Metrics and laws of software evolution-the nineties view, *Proceedings Fourth International Software Metrics Symposium*, IEEE, pp. 20–32.
- Lethbridge, T. C., Sim, S. E. and Singer, J.: 2005, Studying software engineers: Data collection techniques for software field studies, *Empirical software engineering* **10**(3), 311–341.
- Letouzey, J.-L.: 2012, The sqale method for evaluating technical debt, *2012 Third International Workshop on Managing Technical Debt (MTD)*, IEEE, pp. 31–36.
- Letouzey, J.-L. and Ilkiewicz, M.: 2012, Managing technical debt with the sqale method, *IEEE software* **29**(6), 44–51.
- Li, W. and Henry, S.: 1993, Object-oriented metrics that predict maintainability, *Journal of systems and software* **23**(2), 111–122.
- Li, Z., Avgeriou, P. and Liang, P.: 2015, A systematic mapping study on technical debt and its management, *Journal of Systems and Software* **101**, 193–220.
- Lungu, M. F.: 2009, *Reverse engineering software ecosystems*, PhD thesis, Università della Svizzera italiana.
- Lungu, M., Malnati, J. and Lanza, M.: 2009, Visualizing gnome with the small project observatory, *2009 6th IEEE International Working Conference on Mining Software Repositories*, IEEE, pp. 103–106.
- Maldonado, E. d. S., Abdalkareem, R., Shihab, E. and Serebrenik, A.: 2017, An empirical study on the removal of self-admitted technical debt, *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, pp. 238–248.
- Manikas, K.: 2016, Revisiting software ecosystems research: A longitudinal literature study, *Journal of Systems and Software* **117**, 84–103.
- Marantos, C., Lamprakos, C. P., Tsoutsouras, V., Siozios, K. and Soudris, D.: 2018, Towards plug&play smart thermostats inspired by reinforcement learning, *Proceedings of the Workshop on INTelligent Embedded Systems Architectures and Applications*, pp. 39–44.
- March, S. T. and Smith, G. F.: 1995, Design and natural science research on information technology, *Decision support systems* **15**(4), 251–266.
- Martin, R. C.: 2009, *Clean code: a handbook of agile software craftsmanship*, Pearson Education.

- Martini, A., Besker, T. and Bosch, J.: 2018, Technical debt tracking: Current state of practice: A survey and multiple case study in 15 large organizations, *Science of Computer Programming* **163**, 42–61.
- Martini, A., Bosch, J. and Chaudron, M.: 2015, Investigating architectural technical debt accumulation and refactoring over time: A multiple-case study, *Information and Software Technology* **67**, 237–253.
- Mens, T. and Tourwé, T.: 2004, A survey of software refactoring, *IEEE Transactions on software engineering* **30**(2), 126–139.
- Murphy-Hill, E., Parnin, C. and Black, A. P.: 2011, How we refactor, and how we know it, *IEEE Transactions on Software Engineering* **38**(1), 5–18.
- Nikolaidis, N., Digkas, G., Ampatzoglou, A. and Chatzigeorgiou, A.: 2019, Reusing code from stackoverflow: the effect on technical debt, *45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA'19)*, IEEE.
- Olbrich, S., Cruzes, D. S., Basili, V. and Zazworka, N.: 2009, The evolution and impact of code smells: A case study of two open source systems, *2009 3rd international symposium on empirical software engineering and measurement*, IEEE, pp. 390–400.
- Pace, L.: 2012, *Beginning R: An introduction to statistical programming*, Apress.
- Palomba, F., Panichella, A., Zaidman, A., Oliveto, R. and De Lucia, A.: 2017, The scent of a smell: An extensive comparison between textual and structural smells, *IEEE Transactions on Software Engineering* **44**(10), 977–1000.
- Papadopoulos, L., Marantos, C., Digkas, G., Ampatzoglou, A., Chatzigeorgiou, A. and Soudris, D.: 2018, Interrelations between software quality metrics, performance and energy consumption in embedded applications, *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*, pp. 62–65.
- Parnas, D. L.: 1994, Software aging, *Proceedings of 16th International Conference on Software Engineering*, IEEE, pp. 279–287.
- Pérez, B., Brito, J. P., Astudillo, H., Correal, D., Rios, N., Spínola, R. O., Mendonça, M. and Seaman, C.: 2019, Familiarity, causes and reactions of software practitioners to the presence of technical debt: a replicated study in the chilean software industry, *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*, IEEE, pp. 1–7.
- Perry, D. E. and Wolf, A. L.: 1992, Foundations for the study of software architecture, *ACM SIGSOFT Software engineering notes* **17**(4), 40–52.

- Peters, R. and Zaidman, A.: 2012, Evaluating the lifespan of code smells using software repository mining, *2012 16th European Conference on Software Maintenance and Reengineering*, IEEE, pp. 411–416.
- Potvin, R. and Levenberg, J.: 2016, Why google stores billions of lines of code in a single repository, *Communications of the ACM* **59**(7), 78–87.
- Ramac, R., Mandić, V., Tausan, N., Rios, N., de Mendonca Neto, M. G., Seaman, C. and Spínola, R. O.: 2020, Common causes and effects of technical debt in serbian it: Insightd survey replication, *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, IEEE, pp. 354–361.
- Rausch, T., Hummer, W., Leitner, P. and Schulte, S.: 2017, An empirical analysis of build failures in the continuous integration workflows of java-based open-source software, *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, IEEE, pp. 345–355.
- Riaz, M., Mendes, E. and Tempero, E.: 2009, A systematic review of software maintainability prediction and metrics, *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, IEEE, pp. 367–377.
- Rios, N., Spínola, R. O., Mendonça, M. and Seaman, C.: 2018, The most common causes and effects of technical debt: first results from a global family of industrial surveys, *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 1–10.
- Robbes, R., Lungu, M. and Röthlisberger, D.: 2012, How do developers react to api deprecation? the case of a smalltalk ecosystem, *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pp. 1–11.
- Runeson, P., Host, M., Rainer, A. and Regnell, B.: 2012, *Case study research in software engineering: Guidelines and examples*, John Wiley & Sons.
- Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L. and Jaspan, C.: 2018, Lessons from building static analysis tools at google, *Communications of the ACM* **61**(4), 58–66.
- Sas, D., Avgeriou, P. and Fontana, F. A.: 2019, Investigating instability architectural smells evolution: an exploratory case study, *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, pp. 557–567.
- Schermann, G., Cito, J., Leitner, P. and Gall, H. C.: 2016, Towards quality gates in continuous delivery and deployment, *2016 IEEE 24th international conference on program comprehension (ICPC)*, IEEE, pp. 1–4.

- Seaman, C. and Guo, Y.: 2011, Chapter 2—measuring and monitoring technical debt, vol. 82.
- Siavvas, M. and Gelenbe, E.: 2019, Optimum checkpointing for long-running programs, *15th China-Europe International Symposium on software engineering education*.
- Siavvas, M., Gelenbe, E., Kehagias, D. and Tzouvaras, D.: 2018, Static analysis-based approaches for secure software development, *International ISCIS Security Workshop*, Springer, Cham, pp. 142–157.
- Siavvas, M., Marantos, C., Papadopoulos, L., Kehagias, D., Soudris, D. and Tzouvaras, D.: 2019, On the relationship between software security and energy consumption, *15th China-Europe International Symposium on Software Engineering Education*.
- Siavvas, M., Tsoukalas, D., Jankovic, M., Kehagias, D., Chatzigeorgiou, A., Tzouvaras, D., Anicic, N. and Gelenbe, E.: 2019, An empirical evaluation of the relationship between technical debt and software security, *9th International Conference on Information Society and Technology (ICIST)*, Vol. 2019.
- Siegmund, J., Kästner, C., Liebig, J., Apel, S. and Hanenberg, S.: 2014, Measuring and modeling programming experience, *Empirical Software Engineering* **19**(5), 1299–1334.
- Skiada, P., Ampatzoglou, A., Arvanitou, E.-M., Chatzigeorgiou, A. and Stamelos, I.: 2018, Exploring the relationship between software modularity and technical debt, *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, IEEE, pp. 404–407.
- Sommerville, I.: 2011, Software engineering 9th edition, *ISBN-10 137035152*, 18.
- Stroggylos, K. and Spinellis, D.: 2007, Refactoring—does it improve software quality?, *Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007)*, IEEE, pp. 10–10.
- Suryanarayana, G., Sharma, T. and Samarthayam, G.: 2015, Software process versus design quality: Tug of war?, *IEEE Software* (4), 7–11.
- Tan, J., Lungu, M. and Avgeriou, P.: 2018, Towards studying the evolution of technical debt in the python projects from the apache software ecosystem., *BENEVOL*, pp. 43–45.
- Tichy, W. F. and Padberg, F.: 2007, Empirical methods in software engineering research, *29th International Conference on Software Engineering (ICSE'07 Companion)*, IEEE, pp. 163–164.
- Tourani, P., Jiang, Y. and Adams, B.: 2014, Monitoring sentiment in open source mailing lists: exploratory study on the apache ecosystem., *CASCON*, Vol. 14, pp. 34–44.

- Tsimpourlas, F., Papadopoulos, L., Bartsokas, A. and Soudris, D.: 2018, A design space exploration framework for convolutional neural networks implemented on edge devices, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **37**(11), 2212–2221.
- Tsintzira, A.-A., Ampatzoglou, A., Matei, O., Ampatzoglou, A., Chatzigeorgiou, A. and Heb, R.: 2019, Technical debt quantification through metrics: an industrial validation, *15th China-Europe International Symposium on Software Engineering Education (CEISEE 2019)*. IEEE.
- Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A. and Poshyvanyk, D.: 2016, An empirical investigation into the nature of test smells, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 4–15.
- Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A. and Poshyvanyk, D.: 2015, When and why your code starts to smell bad, *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1, IEEE, pp. 403–414.
- Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A. and Poshyvanyk, D.: 2017, When and why your code starts to smell bad (and whether the smells go away), *IEEE Transactions on Software Engineering* **43**(11), 1063–1088.
- Van Vliet, H., Van Vliet, H. and Van Vliet, J.: 2008, *Software engineering: principles and practice*, Vol. 13, John Wiley & Sons Hoboken, NJ.
- Vasilescu, B., Serebrenik, A., Goeminne, M. and Mens, T.: 2014, On the variation and specialisation of workload—a case study of the gnome ecosystem community, *Empirical Software Engineering* **19**(4), 955–1008.
- Vassallo, C., Palomba, F., Bacchelli, A. and Gall, H. C.: 2018, Continuous code quality: are we (really) doing that?, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 790–795.
- Wieringa, R.: 2010, Design science methodology: principles and practice, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pp. 493–494.
- Wilking, D., Kahn, U. F. and Kowalewski, S.: 2007, An empirical evaluation of refactoring, *e-Infomatica Software Engineering Journal* **1**(1).
- Wittern, E., Suter, P. and Rajagopalan, S.: 2016, A look at the dynamics of the javascript package ecosystem, *Proceedings of the 13th International Conference on Mining Software Repositories*, pp. 351–361.

- Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B. and Wesslén, A.: 2012, *Experimentation in software engineering*, Springer Science & Business Media.
- Xing, Z. and Stroulia, E.: 2005, Analyzing the evolutionary history of the logical design of object-oriented software, *IEEE Transactions on Software Engineering* **31**(10), 850–868.
- Xygkis, A., Soudris, D., Papadopoulos, L., Yous, S. and Moloney, D.: 2018, Efficient winograd-based convolution kernel implementation on edge devices, *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, IEEE, pp. 1–6.
- Yamashita, A. and Moonen, L.: 2013, Do developers care about code smells? an exploratory survey, *2013 20th working conference on reverse engineering (WCRE)*, IEEE, pp. 242–251.
- Yin, R. K.: 2017, *Case study research and applications: Design and methods*, Sage publications.
- Yli-Huumo, J., Maglyas, A. and Smolander, K.: 2016, How do software development teams manage technical debt?—an empirical study, *Journal of Systems and Software* **120**, 195–218.
- Zabardast, E., Gonzalez-Huerta, J. and Smite, D.: 2020, Refactoring, bug fixing, and new development effect on technical debt: An industrial case study, *46th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2020)*, IEEE.
- Zazworka, N., Shaw, M. A., Shull, F. and Seaman, C.: 2011, Investigating the impact of design debt on software quality, *Proceedings of the 2nd Workshop on Managing Technical Debt*, pp. 17–23.