University of Groningen

Technical Debt Repayment in Practice

Tan, Jie

DOI:
[10.33612/diss.195698574](https://doi.org/10.33612/diss.195698574)

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):

Tan, J. (2021). *Technical Debt Repayment in Practice*. [Thesis fully internal (DIV), University of Groningen]. University of Groningen. <https://doi.org/10.33612/diss.195698574>

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.

university of
groningen

Technical Debt Repayment in Practice

PhD thesis

to obtain the degree of PhD at the
University of Groningen
on the authority of the
Rector Magnificus Prof. C. Wijmenga
and in accordance with
the decision by the College of Deans.

This thesis will be defended in public on
Friday 17 December 2021 at 16.15 hours

by

Jie Tan

born on 15 December 1991
in Hebei, China

Supervisor

Prof. P. Avgeriou

Co-supervisor

Dr. D. Feitosa

Assessment committee

Prof. T. van der Storm

Prof. A. Chatzigeorgiou

Prof. F. Arcelli Fontana

The research reported in this thesis has been conducted in the Software Engineering and Architecture group of the Bernoulli Institute for Mathematics, Computer Science and Artificial Intelligence of the University of Groningen, The Netherlands.

Technical Debt Repayment in Practice
Jie Tan

To my parents

谨以此书献给我敬爱的父母

Abstract

Technical Debt (TD) is a metaphor used to describe a trade-off between the short-term benefits of “cutting corners” in software development and the long-term sustainability of a software system. Researchers have classified technical debt into several different types, such as design or code debt. The amount of technical debt tends to increase during the evolution of software systems. When the increasing technical debt results in extra maintenance and evolution effort, it needs to be (at least partially) managed.

Technical debt repayment is the TD management activity which refers to the remediation of technical debt by reworking design and implementation constructs. Paying back all technical debt may be infeasible since there is typically a shortage of resources that can be allocated. Moreover, business stakeholders often prioritize resources (money and time) on developing new features and fixing defects, rather than performing TD refactoring activities. Therefore, developers need to improve resource allocation for TD repayment due to the limited resources and competing business interests. Thus, investigating the factors that influence TD repayment is crucial in finding optimal solutions that maximize value and satisfy resource constraints. **The thesis reports on the investigation of how TD is paid back in practice and what factors influence this repayment, by considering both software repositories and the perspective of practitioners.**

In order to address the stated problem, we first looked into how TD repayment occurs in practice and its evolution over time. Specifically, we focused on the TD fixing rate across projects and across issue types, the repayment effort and the survival time of TD items. The results showed that most of the repayment effort goes into improving testing, adding documentation, reducing complexity, and removing duplication. We also extended a previous study on the same topic that used several Java projects and compared the results of TD remediation in Python and Java

projects. The results indicated that biases such as project-dependent variables (complexity, domain, etc.) might affect the comparison between the programming languages. Analyzing TD remediation can help researchers gain a better understanding of the overview of TD evolution and obtain a preliminary knowledge of how developers manage TD.

During the investigation of technical debt remediation, we found that some TD types frequently co-occur, which poses a great risk as it can intensify the negative consequences caused by individual TD items. Although co-occurring TD has been previously investigated, there is little work beyond the co-occurrence of code smells. In addition, none of the current research examines the time interval between co-occurring issues' appearance and disappearance; the survival time of co-occurring issues can provide insights on how they are repaid in practice. Thus, we conducted a case study to investigate which types of TD frequently co-occur and how long is the interval between their appearance and disappearance. The results showed that the phenomenon of co-occurring TD is highly spread, and Design Debt issues more commonly co-occur with issues of similar nature. In addition, co-occurring issues tend to be introduced within the same week. However, after one of the co-occurring issues is removed, the second one survives for much longer (often around a year). Moreover, co-occurring Design Debt issues seem to be paid back in a shorter time, while most of the Test Debt issues are introduced before the other debt types but are also the last to be repaid.

Another observation that stemmed from the investigation of technical debt remediation was that a considerable portion of the TD items were fixed by the same developers who introduced them; we call this *self-fixed technical debt*. Investigating self-fixed technical debt can shed light on how and why developers prioritize the remediation of different types of TD, either their own or of others. Hence, we looked into the phenomenon of self-fixed technical debt, especially how does self-fixing influence the prioritization of TD repayment. As this is a relatively new concept, we conducted two empirical studies (case studies) to investigate it.

In the first case study, we analyzed how much TD is self-fixed, which types of TD are more likely to be self-fixed and how long self-fixed TD survives from the software repositories. The results showed that about half of the fixed TD issues were paid back by the same developers who introduced them. Moreover, the projects that are larger and have more commits and TD issues tend to have a relatively lower likelihood of observing self-fixing. Furthermore, there is no significant difference in the survival time of self-fixed and non-self-fixed issues. Looking at the types of TD, Defect Debt and Code Debt are more likely to be self-fixed than other debt types. Moreover, we observed that developers who are more dedicated to or knowledgeable of the project (e.g., by contributing more commits) may drive the

overall self-fixing up. Finally, the longer the developers are involved in a project, the less often they self-fix TD.

The answer to the above first case study provides valuable insights, but only by analyzing source code, not based on the opinion of software developers. Consequently, there is no evidence of whether developers consider the differences between various debt types when repaying their own debt and whether they take other factors into account. Thus, the second case study is to investigate the phenomenon of self-fixed TD from the perspective of practitioners. To this end, we conducted a survey with 17 well-known Java and Python open-source software communities to consider the practitioners' intent and experiences. The results showed that most practitioners display a positive attitude towards self-fixing TD, and seem mostly attentive to Code Debt. Moreover, when looking into the reasons to introduce and then self-fix TD, we found that non-technical reasons (e.g., related to planning and management) are the most common to justify both introducing and self-fixing TD, and are often mentioned together with technical reasons (e.g., related to the development process). In particular, many developers mentioned a sense of responsibility as a factor for self-fixing, and that repayment decisions are not made easily but by balancing costs and benefits, among other factors.

Having investigated TD remediation both from source code analysis and from the perspective of developers in the previous research questions, we realized that we are only looking at the end result, instead of the process that led to the result. In other words, we are missing the full picture of TD repayment: from the time a TD item is incurred in a commit until it is fixed with another commit. This wider perspective cannot be thoroughly examined by looking at a single data source (e.g., source code or developers' conversations). Thus, we looked at two data sources (issue trackers and source code) to investigate the timeline of TD items incurred in source code, reported and discussed in issue trackers and then paid back in source code. For that, we manually extracted and inspected 219 TD items from five open-source projects.

The results showed that most of the TD items identified and mentioned as resolved in issue trackers are also paid back in source code, while about one-fifth are discussed further after being identified. The resolved TD items in issue trackers related to Design Debt have a significantly higher chance of being paid back in source code, while Test Debt items are less likely. Furthermore, although TD items take a long time to be identified (around one year), they tend to be resolved in source code within a few days after that. Also, TD items seem to be more quickly identified in the issue tracker by the same developers who introduced them in the source code. After TD items are identified in the issue tracker, they are also more quickly resolved in the source code by the same developers who identified them.

Samenvatting

Technical Debt (Technische schuld-TD) is een metafoor die wordt gebruikt om de afweging te beschrijven tussen de kortetermijnvoordelen van het “afsnijden van bochten” bij softwareontwikkeling en de duurzaamheid van een softwaresysteem op lange termijn. Onderzoekers hebben technische schuld in verschillende soorten ingedeeld, bijvoorbeeld ontwerp of code schuld. De omvang van de technische schuld heeft de neiging toe te nemen tijdens de evolutie van softwaresystemen. Wanneer de toenemende technische schuld resulteert in extra onderhouds- en evolutie-inspanningen, moet deze (ten minste gedeeltelijk) worden beheerd.

Het aflossen van de technische schuld is de TD-managementactiviteit die verwijst naar het verhelpen van de technische schuld door het herwerken van ontwerp- en implementatieconstructies. Het terugbetalen van alle technische schulden kan onhaalbaar zijn omdat er meestal een tekort is aan middelen die kunnen worden toegewezen. Bovendien geven zakelijke belanghebbenden vaak voorrang aan middelen (geld en tijd) voor het ontwikkelen van nieuwe functies en het repareren van defecten, in plaats van het uitvoeren van TD refactoring activiteiten. Daarom moeten ontwikkelaars de toewijzing van middelen voor TD refactoring verbeteren vanwege de beperkte middelen en concurrerende zakelijke belangen. Dus is het onderzoeken van de factoren die TD terugbetaling beïnvloeden van cruciaal belang bij het vinden van optimale oplossingen die waarde maximaliseren en voldoen aan de beperkingen van middelen. **Het proefschrift doet verslag van het onderzoek naar hoe TD in de praktijk wordt terugbetaald en welke factoren deze terugbetaling beïnvloeden, door zowel software repositories als het perspectief van praktijkmensen te beschouwen.**

Om dit probleem aan te pakken hebben wij eerst onderzocht hoe de terugbetaling van TD in de praktijk verloopt en hoe dit in de loop van de tijd evolueert. In het bijzonder hebben we ons gericht op het vastleggingspercentage van TD tussen

projecten en probleemtypes, de terugbetalingsinspanning en de overlevingstijd van TD-items. De resultaten toonden aan dat het grootste deel van de terugbetalingsinspanning gaat naar het verbeteren van testen, het toevoegen van documentatie, het verminderen van complexiteit en het verwijderen van duplicatie. We hebben ook een eerdere studie over hetzelfde onderwerp, waarin verschillende Java-projecten werden gebruikt, uitgebreid en de resultaten van TD-reparatie in Python- en Java-projecten vergeleken. De resultaten gaven aan dat biases zoals projectafhankelijke variabelen (complexiteit, domein, enz.) de vergelijking tussen de programmeertalen kunnen beïnvloeden. Het analyseren van TD sanering kan onderzoekers helpen een beter inzicht te krijgen in het overzicht van TD evolutie en een voorlopige kennis te verkrijgen van hoe ontwikkelaars TD beheren.

Tijdens het onderzoek naar de remediëring van technische schuld, ontdekten we dat sommige TD types vaak samen voorkomen, wat een groot risico inhoudt omdat het de negatieve gevolgen van individuele TD items kan versterken. Hoewel co-occurring TD eerder is onderzocht, is er weinig werk dat verder gaat dan de co-occurrence van code smells. Bovendien is er in geen van de huidige onderzoeken gekeken naar het tijdsinterval tussen het verschijnen en verdwijnen van een aantal vormen van TD; de overlevingsduur van een aantal vormen van TD kan inzicht verschaffen in de wijze waarop deze in de praktijk worden afgelost. Daarom hebben we een casestudy uitgevoerd om na te gaan welke soorten TD vaak samen voorkomen en hoe lang het interval is tussen het verschijnen en verdwijnen ervan. Uit de resultaten bleek dat het fenomeen van de samenvallende TD sterk verspreid is, en dat problemen in verband met ontwerpschulden vaker samenvallen met problemen van soortgelijke aard. Bovendien worden samen voorkomende problemen vaak binnen dezelfde week geïntroduceerd. Nadat een van de samenkomende problemen is opgelost, blijft het tweede probleem echter veel langer bestaan (vaak ongeveer een jaar). Bovendien worden samen voorkomende ontwerpschulden in kortere tijd terugbetaald, terwijl de meeste testschulden eerder worden ingevoerd dan de andere soorten schulden, maar ook het laatst worden terugbetaald.

Een andere vaststelling die voortvloeyde uit het onderzoek naar het wegwerken van de technische schuld, was dat een aanzienlijk deel van de TD-items werd hersteld door dezelfde ontwikkelaars die ze hadden ingevoerd; wij noemen dit *zelfgefixeerde technische schuld*. Onderzoek naar zelfgefixeerde technische schuld kan licht werpen op hoe en waarom ontwikkelaars prioriteit geven aan het herstellen van verschillende soorten TD, hetzij hun eigen of van anderen. Daarom hebben we het fenomeen van zelfgefixeerde technische schuld onderzocht, in het bijzonder hoe zelffixatie de prioritering van TD-terugbetaling beïnvloedt. Omdat dit een relatief nieuw concept is, hebben we twee empirische studies (case studies) uitgevoerd om het te onderzoeken.

In de eerste casestudy hebben we geanalyseerd hoeveel TD er zelf hersteld wordt, welke soorten TD meer kans hebben om zelf hersteld te worden en hoe lang zelf herstelde TD overleeft uit de software repositories. De resultaten toonden aan dat ongeveer de helft van de opgeloste TD problemen werden terugbetaald door dezelfde ontwikkelaars die ze introduceerden. Bovendien hebben projecten die groter zijn en meer commits en TD-problemen hebben, een relatief kleinere kans op het waarnemen van zelf-oplossing. Verder is er geen significant verschil in de overlevingstijd van zelf-vastgestelde en niet-zelf-vastgestelde problemen. Als we kijken naar de soorten TD, dan zien we dat Defect Debt en Code Debt vaker zelf worden opgelost dan andere soorten schuld. Bovendien hebben we waargenomen dat ontwikkelaars die meer toegewijd zijn aan of meer kennis hebben van het project (bijvoorbeeld door meer commits bij te dragen) de totale zelf-oplossing kunnen opdrijven. Tenslotte, hoe langer de ontwikkelaars betrokken zijn bij een project, hoe minder vaak ze TD zelf oplossen.

Het antwoord op de bovenstaande eerste casestudy biedt waardevolle inzichten, maar analyseert alleen de broncode en kijkt niet naar de mening van softwareontwikkelaars. Bijgevolg is niet bewezen of ontwikkelaars bij het terugbetalen van hun eigen schuld rekening houden met de verschillen tussen de verschillende soorten schulden en of zij andere factoren in aanmerking nemen. De tweede casestudy is dus om het fenomeen van zelf-vastgestelde TD te onderzoeken vanuit het perspectief van de beoefenaars. Daartoe hebben we een enquête gehouden onder 17 bekende Java en Python open-source software gemeenschappen om de intentie en ervaringen van de ontwikkelaars te onderzoeken. De resultaten toonden aan dat de meeste ontwikkelaars een positieve houding vertonen ten opzichte van zelfherstellende TD, en vooral aandacht lijken te hebben voor Code Debt. Bovendien, wanneer we kijken naar de redenen om TD op te bouwen en vervolgens zelf af te lossen, vinden we dat niet-technische redenen (bv. gerelateerd aan planning en management) het meest voorkomen om zowel TD op te bouwen als het zelf aflossen, en vaak samen met technische redenen (bv. gerelateerd aan het ontwikkelproces) worden genoemd. Veel ontwikkelaars noemden met name verantwoordelijkheidsgevoel als factor voor zelfoplossing, en dat beslissingen over terugbetaling niet gemakkelijk worden genomen, maar door kosten en baten tegen elkaar af te wegen, naast andere factoren.

Na het onderzoeken van TD remediation zowel vanuit broncode analyse als vanuit het perspectief van ontwikkelaars in de vorige onderzoeksvragen, realiseerden we ons dat we alleen kijken naar het eindresultaat, in plaats van naar het proces dat tot het resultaat leidde. Met andere woorden, we missen het volledige plaatje van TD terugbetaling: vanaf het moment dat een TD item wordt opgelopen in een commit tot het moment dat het wordt opgelost met een andere commit. Dit bredere

perspectief kan niet grondig onderzocht worden door naar een enkele gegevensbron te kijken (bijv. broncode of gesprekken van ontwikkelaars). Daarom hebben we gekeken naar twee gegevensbronnen (issue trackers en broncode) om de tijdlijn te onderzoeken van TD items die ontstaan in broncode, gerapporteerd en besproken worden in issue trackers en vervolgens terugbetaald worden in broncode. Daarvoor hebben we 219 TD-items uit vijf open-sourceprojecten handmatig geëxtraheerd en geïnspecteerd.

De resultaten toonden aan dat de meeste TD-items die in issue trackers geïdentificeerd en als opgelost vermeld worden, ook terugbetaald worden in broncode, terwijl ongeveer een vijfde verder besproken wordt nadat ze geïdentificeerd zijn. De opgeloste TD items in issue trackers gerelateerd aan Design Debt hebben een significant hogere kans om terugbetaald te worden in broncode, terwijl Test Debt items minder kans hebben. Bovendien, hoewel het lang duurt om TD items te identificeren (ongeveer een jaar), hebben ze de neiging om binnen een paar dagen daarna opgelost te zijn in broncode. Ook lijken TD-items sneller in de issue tracker geïdentificeerd te worden door dezelfde ontwikkelaars die ze in de broncode hebben geïntroduceerd. Nadat TD items zijn geïdentificeerd in de issue tracker, worden ze ook sneller opgelost in de broncode door dezelfde ontwikkelaars die ze hebben geïdentificeerd.

Contents

Abstract

Samenvatting

List of Figures **v**

List of Tables **vii**

Acknowledgements **ix**

1 Introduction **1**

1.1 The concept of Technical Debt 1

1.2 Technical Debt Management and Evolution 2

1.3 Technical Debt in Java vs. Python 2

1.4 Research Design 3

1.4.1 Problem Statement 3

1.4.2 Design Science Framework 5

1.4.3 Problem Decomposition 5

1.4.4 Empirical Research Methodology 8

1.5 Overview of the Dissertation 9

**2 Evolution of Technical Debt Remediation in Python: A Case Study on the
Apache Software Ecosystem** **13**

2.1 Introduction 13

2.2 Study Design 16

2.2.1 Objective and Research Questions 16

2.2.2 Cases Selection 18

2.2.3	Technical Debt Identification	18
2.2.4	Data Analysis	22
2.3	Results	23
2.3.1	Evolution of Technical Debt Remediation in Python & Comparison to Java	24
2.3.2	Learning from the Differences between Python and Java	36
2.4	Discussion	39
2.4.1	Common Rules in Both Languages	40
2.4.2	Implications	43
2.5	Threats to Validity	45
2.5.1	Construct Validity	45
2.5.2	External Validity	46
2.5.3	Reliability	46
2.5.4	Confounding Factors	46
2.6	Related Work	47
2.7	Conclusion	48
3	Investigating the Relationship between Co-occurring Technical Debt in Python	51
3.1	Introduction	51
3.2	Study Design	53
3.2.1	Objectives and Research Questions	54
3.2.2	Case Selection	54
3.2.3	Variables	55
3.2.4	Data Analysis	56
3.3	Results	59
3.3.1	RQ1: Which types of technical debt frequently co-occur?	59
3.3.2	RQ2: How long is the interval between the appearance and disappearance of co-occurring technical debt issues?	62
3.4	Discussion	63
3.4.1	Interpretation of Results	63
3.4.2	Implications for Researchers and Practitioners	64
3.5	Threats to Validity	65
3.6	Related Work	66
3.7	Conclusion	67
4	Does it matter who pays back Technical Debt? An empirical study of self-fixed TD	69
4.1	Introduction	70

Contents

4.2	Related work	72
4.3	Study design	74
4.3.1	Objective and Research Questions	74
4.3.2	Case Selection	75
4.3.3	Variables and Data Collection	77
4.3.4	Data Analysis	79
4.4	Results	81
4.4.1	How much technical debt is self-fixed in open source projects?	81
4.4.2	Which type of technical debt is more likely to be self-fixed?	83
4.4.3	How long does self-fixed technical debt survive during the evolution of a system?	85
4.4.4	How does the number of developers and the frequency of their modifications relate with the likelihood of TD issues to be self-fixed?	88
4.4.5	Who (self) fixes technical debt?	90
4.5	Discussion	91
4.5.1	Interpretation of Results	92
4.5.2	Implications for Researchers and Practitioners	94
4.6	Threats to Validity	96
4.7	Conclusions	99
5	Do practitioners intentionally self-fix Technical Debt and why?	101
5.1	Introduction	101
5.2	Related Work	103
5.3	Research approach	104
5.3.1	Research Questions	104
5.3.2	Survey Design	105
5.3.3	Projects and Participants Selection	107
5.3.4	Data Analysis	108
5.4	Results	110
5.4.1	Do practitioners intentionally self-fix technical debt? (RQ ₁)	111
5.4.2	What reasons motivate practitioners to introduce and self-fix TD? (RQ ₂)	113
5.4.3	How do human factors influence practitioners in self-fixing TD? (RQ ₃)	116
5.5	Discussion	121
5.5.1	Interpretation of Results	121
5.5.2	Implications to Researchers and Practitioners	123
5.6	Threats to Validity	123

5.7	Conclusions	124
6	Complete Debt Stories as Told by Source Code and Issue Tracker: An Empirical Study of Technical Debt Timelines	127
6.1	Introduction	127
6.2	Related Work	130
6.3	Study Design	131
6.3.1	Objectives and Research Questions	131
6.3.2	Project Data Extraction	133
6.3.3	Variables and Data Collection	134
6.3.4	Data Analysis	137
6.4	Results	139
6.4.1	To what extent technical debt items identified and discussed in issue trackers are paid back in source code? (RQ1)	139
6.4.2	How long are the time intervals between introducing, identifying, discussing and paying back TD items? (RQ2)	140
6.4.3	How can the number and involvement of developers associate with the management of technical debt items documented in issue trackers? (RQ3)	143
6.5	Discussion	148
6.5.1	Interpretation of Results	148
6.5.2	Implications for Researchers and Practitioners	149
6.6	Threats to Validity	149
6.7	Conclusions	150
7	Conclusions and Future Work	151
7.1	Research Questions and Contributions	151
7.2	Future Work	155
7.2.1	Exploration of Additional Programming Languages	155
7.2.2	Human Factors	155
7.2.3	Industrial Applicability	155
	Appendix A	157
A.1	Supplementary Tables to Chapter 2	157
	Appendix B	161
B.1	Supplementary Table to Chapter 5	161
	Bibliography	165

List of Figures

1.1	Problem decomposition overview	6
2.1	Percentage of fixed issues (black) in each project.	24
2.2	Distribution of fixing rate for small and large number of total commits	25
2.3	Distribution of fixed issues for the 56 rules detected in this study. Every bar represents a rule, height proportional to the number of fixed issues	26
2.4	Survival time histogram of 43K fixed issues with bins of 30 days	34
2.5	Distribution of median survival time for 20 rules with the highest fixing rates (according to RQ _{1.3})	35
3.1	Network of top 10% co-occurring rule violations	59
3.2	Distribution of the possibility of co-occurrence and lift values for the violations of 29 pairs of rules	60
3.3	Distribution of the possibility of co-occurrence and lift values for the violations of 27 pairs of rules	62
4.1	Distribution of self-fixing rates of five TD types among 20 Python and 16 Java projects	84
4.2	Results of Kaplan-Meier method for the survival time of self-fixed and non-self-fixed issues	86
4.3	Results of Kaplan-Meier method for the survival time for self-fixed and non-self-fixed issues belonging to different types	87
5.1	Design of the survey in our study.	106
5.2	Respondents' knowledge level of technical debt	111

5.3	Distribution of self-fixed TD types	112
5.4	Percentage of participants indicating the frequency of self-fixing TD (column) for a given frequency of TD introduction (row).	113
5.5	Reasons to introduce technical debt	114
5.6	Reasons to self-fix technical debt	116
5.7	Association between reasons	117
5.8	The percentages of how often participants in each demographic group have self-fixed TD	118
5.9	The percentages of self-fixing certainty levels for participants in each demographic group	119
6.1	TD timeline in issue trackers and source code.	129
6.2	TD timeline	141
6.3	Distribution of Time Intervals between Introducing, Identifying, Dis- cussing and Paying Back All TD Items	142
6.4	Distribution of Time Intervals between Introducing, Identifying, Dis- cussing and Paying Back Different Types of TD Items	143
6.5	Results of Kaplan-Meier method for the time interval 1 of all the 219 TD items	144
6.6	Results of Kaplan-Meier method for the time interval 3a of all the 41 TD items	146
6.7	Results of Kaplan-Meier method for the time interval 3b of all the 219 TD items	147

List of Tables

1.1	Overview of research methodology	9
1.2	Overview of dissertation	10
2.1	Migration-related issues	21
2.2	Number of fixed and open issues for priority type	22
2.3	Stratified random sampling for 426 issues	27
2.4	Fixing Prevalence for Top 15 Rules	28
2.5	Fixing Rate for Rules with at Least 500 Occurrences	30
2.6	Remediation Effort Distribution: Ten issues are responsible for more than 90% of the remediation effort	33
2.7	Comparison of 15 common rules for Python and Java	41
3.1	Mapping of highlighted rules to types of technical debt	61
4.1	Details of the 20 Python and 16 Java Projects	81
4.2	Results of the generalized linear mixed model	83
4.3	Results of Wilcoxon Rank Sum tests and Cliff's Delta Effect Size	84
4.4	Ranks of five debt types according to the Scott-Knott ESD tests	85
4.5	Percentages of self-fixed and non-self-fixed issues within different time periods	86
4.6	Ranks of five debt types according to the Scott-Knott ESD tests	88
4.7	Results of the generalized linear mixed model	89
4.8	Results of the generalized linear mixed model	91
5.1	TD general questions of the survey	107
5.2	Mapping between survey questions, variables and research questions	108

5.3	Characteristics of the respondents	110
6.1	Details of the Five Selected Projects	134
6.2	Different TD Types that are Detected in Issue Trackers	136
6.3	Issue and Technical Debt Details of the Five Selected Projects	137
6.4	Number of Issues per TD Type	137
6.5	Numbers and Percentages of TD Items Resolved in Issue Trackers (IT) are Paid Back in Source Code (SC)	141
6.6	Results of the generalized linear mixed model	145
7.1	Contributions of the PhD dissertation	152
A.2	Project statistics	158
A.3	Issue rules	159
B.4	Self-fixing rates for individual rules	161

Acknowledgements

My PhD journey began in the early spring of 2018 in the Netherlands, a romantic country with tulips and windmills. Now it is time to conclude my nearly four-year life in this lovely place and express my gratitude to all the people involved in my life during this period.

First and foremost, I would like to express my utmost gratitude to my supervisors, Prof. Paris Avgeriou and Dr. Daniel Feitosa. Dear Prof. Paris Avgeriou, I am amazed at your profound knowledge and have learned a lot from you. Not only doing research and writing papers, but more importantly, the scientific way of thinking will benefit me for my whole life. I am also eternally grateful that you always help me continue my research and manage my PhD at critical moments. Dear Dr. Daniel Feitosa, you have always been patient, optimistic and encouraging. When I confronted some difficulties in my research, you were always on my side, supported me and gave me timely help. No matter how busy you were, you always gave me meticulous supervision and spent a lot of time on my research, which I really appreciate. In the past few years, I have enjoyed our scientific discussions and casual conversations so much. I am incredibly fortunate to have you as my co-supervisor!

I would like to express my thanks to the members of the assessment committee, Prof. Tijs van der Storm, Prof. Alexander Chatzigeorgiou and Prof. Francesca Arcelli Fontana, for your time and effort in reading my thesis and making valuable comments.

I am also grateful to Darius Daniel Sas and Sha Luo for accepting to be my paranympths. Darius, we started our PhD life at about the same time, and we have made progress and experienced a lot together in the past four years. You are a great partner for working, sharing and traveling, making my life in the Netherlands more colorful and exciting. Sha, we shared a lot of incredible memories together and you have been the best friend I could wish for! I wish you a lot of success in your future

career.

I would like to thank all other members of the Software Engineering and Architecture (SEARCH) group: Dr. Andrea Capiluppi, Anja Reuter, Dr. Apostolos Ampatzoglou, Dr. Areti Ampatzoglou, Dr. Ayushi Rastogi, Cezar Sas, Dr. Chen Yang, Christian Manteuffel, Héctor Fabio Cadavid Rengifo, Dr. Georgios Digkas, Gunnar Kudrjavets, Dr. Mohamed A.M. Soliman, Prof. Peng Liang, Ruiyin Li, Sara Mahdavi-Hezavehi, Dr. Sofia Charalampidou, Prof. Tijs van der Storm, Dr. Vasilios Andrikopoulos, Yikun Li, Zaki Pauzi, Dr. Zengyang Li. Many thanks for the support, and I enjoyed a lot from our lunch and coffee breaks, and also all the events we had together. Special thanks to Dr. Mircea Lungu, you are my daily supervisor for the first several months and I appreciate all your help at the beginning of my PhD.

Last but not least, I would like to express my deepest gratitude to my parents for their unconditional and endless support. I am so grateful that my parents always encourage me to explore the world and also proud to have parents like them!

Finally, I wholeheartedly thank all the people encountered in my PhD life.

Jie Tan
Groningen
December 2, 2021

Chapter 1

Introduction

This chapter elaborates on the main concepts of this PhD project, setting the context of the research work and describing the overall research design. We focus on background information regarding technical debt in Section 1.1 and its evolution and management in Section 1.2. We further discuss the possible impact of different programming languages (Python and Java are relevant for this PhD project) on technical debt in Section 1.3. Section 1.4 provides an overview of the problem statement that motivated this work, decomposes that problem into sub-problems that can be solved in individual empirical studies, and subsequently elaborates on the research methods that have been used in the project. Finally, Section 1.5 describes the organization of the remainder of this dissertation.

1.1 The concept of Technical Debt

Technical Debt (TD) is a metaphor used to describe a trade-off between the short-term benefits of “cutting corners” in software development and the long-term sustainability of a software system (Li et al., 2015). Ward Cunningham (Cunningham, 1992) coined this metaphor in 1992 and described it 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.”*

As the focus on technical debt has increased in recent years, several different definitions of this concept have emerged. This thesis relies on one well-accepted definition proposed by Avgeriou et al. (2016) in 2016: *“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.”*

Researchers have further classified technical debt into a number of different

types, depending on the nature of the aforementioned “design or implementation constructs”. For example, Alves et al. (2014) proposed an ontology of 13 types of technical debt, five of which are being widely studied in this thesis: Code Debt (problems found in the source code which can negatively affect making changes), Defect Debt (defects, bugs, or failures found in software systems), Design Debt (technical shortcuts that are taken in detailed design or violations of design principles), Documentation Debt (insufficient, incomplete, or outdated documentation in any aspect of software development) and Test Debt (shortcuts taken in testing such as incomplete code coverage). These five types have also been extensively investigated in other studies (Baldassarre et al., 2020; Digkas et al., 2018; Bavota and Russo, 2016).

1.2 Technical Debt Management and Evolution

When technical debt results in increasingly extra maintenance and evolution effort, it needs to be (at least partially) managed. Technical Debt Management (TDM) comprises a set of activities that prevent potential TD from being incurred, or deal with existing TD to keep it under a reasonable level (Li et al., 2015). Among the nine TDM activities proposed by Li et al. (2015), **TD repayment** is the focus of this thesis and encompasses practices for resolving TD in a software system. Refactoring, design improvements, and test improvements are the most popular TD repayment practices for the cases of Code, Design and Test Debt respectively (Pérez et al., 2021).

During the evolution of software systems, the amount of TD items within those systems increases over time (Chatzigeorgiou and Manakos, 2010). Although a large percentage of TD is paid back during software evolution (Digkas et al., 2018), some items could survive for a long time (Tufano et al., 2017; Potdar and Shihab, 2014). Moreover, the developer who pays back TD is oftentimes the same that introduced it in the system (Bavota and Russo, 2016). Paying back all technical debt may be infeasible since there is typically a shortage of resources that can be allocated. Therefore, it is essential to improve resource allocation for TD repayment, e.g., by determining which TD items should be repaid first and which TD items can be tolerated.

1.3 Technical Debt in Java vs. Python

A large number of studies that analyzed technical debt management have primarily focused on software written in Java (Digkas et al., 2018; Digkas et al., 2020; Li et al., 2020; Palomba et al., 2018b). However, studying the phenomenon of technical debt management, especially TD repayment within a single programming language is

limited in terms of external validity. Thus, this PhD project has taken a second programming language into account, i.e., Python in order to increase the generalizability of the results and also compare to previous work on Java.

Python and Java are two of the most popular object-oriented programming languages in practice. Although both of them are strongly typed languages, Java is statically typed while Python is dynamically typed. Generally, since Python code is more change-prone due to having a higher number of dynamic features (Wang et al., 2015), developers have to spend extra effort on software maintenance (Wang et al., 2015) and software quality improvement (Ray et al., 2014) in Python. In addition, Python code containing dynamic features is inserted or updated more frequently when fixing bugs (Chen et al., 2018). Altogether, the misuse of dynamic features could lead to coding issues that are often fixed by adding tests (Test Debt remediation) or exception handling (Defect Debt remediation) Chen et al. (2018).

Moreover, Python has two major versions (i.e., Python 2 and Python 3) and developers often add workarounds and mix code from the two versions. This kind of backward compatibility results in the usage of redundant and complicated features of the language. Therefore, Python code is characterized by a unique kind of technical debt belonging to Defect Debt. Furthermore, since Java has experienced consistent development in its programming efficiency for decades, general purpose development has been dominated by it and the majority of academic research also focused on it, developers already have some experience to deal with TD in Java projects. However, only few studies have focused on TD in Python projects. Thus, most of the work in this thesis is related to the Python projects, to further investigate the impact of the programming languages' features on TD remediation.

1.4 Research Design

The research project reported in this dissertation aims at tackling a core problem, which is stated in Section 1.4.1. To identify and plan the research activities that are necessary to address this problem, we adopted the Design Science framework as defined by Wieringa (2014); this is described in Section 1.4.2. The breakdown of the core problem into a number of sub-problems is detailed in Section 1.4.3. The last section, 1.4.4, describes the research methods that were used to address those sub-problems.

1.4.1 Problem Statement

Incurring technical debt can bring short-term benefits in terms of time and/or resources, e.g., shorter time to market, less effort required to implement a fea-

ture. However, this expediency has a detrimental effect on the future maintainability and evolvability of the system, as it becomes increasingly harder to make changes (Digkas et al., 2018). This effect translates into additional effort for developers and, subsequently extra costs that can even become prohibitive to organizations. If not repaid, TD can lead to severe quality problems, unexpected cost overruns and substantial financial loss (Tom et al., 2013).

Due to the limited allocated resources for TD repayment, paying back all TD existing in a system is infeasible (Ampatzoglou et al., 2015; Guo et al., 2016; Alfayez et al., 2020; Freire et al., 2020). To support TD repayment, researchers have contributed various approaches, e.g., investigating how particular forms of debt such as self-admitted TD are paid back (Bavota and Russo, 2016; Maldonado et al., 2017) and how repayment relates to software quality metrics such as change-proneness (Palomba et al., 2018b) and maintenance effort (Sjøberg et al., 2013). Nevertheless, business stakeholders often prioritize resources (money and time) on developing new features and fixing defects, rather than performing TD refactoring activities (Martini et al., 2015). Moreover, TD repayment becomes more effort-intensive as the software evolves, since TD accumulates and more components are affected (Digkas et al., 2017); the need to refactor more, larger and more complex components, can in turn discourage repayment and draw stakeholders' attention to more pressing business.

With limited resources and competing business interests, developers need to decide which TD items can be repaid to achieve a higher return of investment, as there is typically a trade-off between the value achieved from repaying a TD item and its cost (Guo et al., 2016). Thus, investigating the factors that influence TD repayment, is crucial in finding optimal solutions that maximize value and satisfy resource constraints. In order to study those factors and potential best practices for TD repayment, we need both hard data from software repositories (e.g., source code, issue trackers) regarding how repayment is done in practice, as well as the perspective of practitioners.

Summarizing the aforementioned points, this thesis addresses the following core problem:

If TD is not repaid, it can accumulate leading to severe quality problems, unexpected cost overruns and substantial financial loss. Due to the limited available resources, development teams need to make well-informed investment decisions on which TD repayment actions will yield the highest return. Thus, it is paramount to investigate how TD is paid back in practice and what factors influence this repayment, by considering both software repositories and the perspective of practitioners.

1.4.2 Design Science Framework

Design Science was introduced by March and Smith (1995), and is a framework that focuses on developing or improving technological solutions for the benefit of stakeholders, who in turn define the scope of the research project. The framework was later refined by Wieringa (2010) to suit software engineering research. The main components of Wieringa's version of the Design Science framework are *practical problems* and *knowledge questions*. The *practical problems* drive the design activities, i.e., they provide the purpose and constraints for the artifact to be designed. The *knowledge questions* steer the activities, i.e., they provide the theories, scientific knowledge, and existing designs that are relevant to the problem being solved.

The *practical problems*, such as the problem statement in this dissertation, are solved by *design cycles*, and encompass three activities: problem investigation, treatment design/identification, and treatment validation. The *knowledge questions* are answered by using *empirical cycles*, such as case studies, experiments and surveys, and thus their activities vary according to the question being asked. The design and empirical cycles are seamlessly nested to develop the artifact that was initially the target of the Design Science project. The aforementioned Design Science workflow is well-suited for describing long-term research such as PhD projects, since it facilitates the decomposition of a problem into practical problems that lead to new practical problems or knowledge questions in an iterative manner.

1.4.3 Problem Decomposition

The decomposition of the problem addressed in this dissertation is depicted in Figure 1.1. The problem statement (see Section 1.4.1) describes the main challenge this dissertation attempts to address, i.e., to investigate how TD is paid back in practice and what factors can influence this repayment. The white boxes represent knowledge questions that were derived based on the Design Science framework; these knowledge questions correspond to the research questions of this dissertation and are numbered as RQ_x (or $RQ_{x,y}$ in case of a decomposed research question). Gray boxes represent empirical cycles, i.e., scientific studies proposed to answer the knowledge questions and that are reported in the individual chapters of this dissertation. Moreover, thick, white arrows denote a sequence, single-line arrows denote decomposition, and dashed arrows denote the investigation of research questions (through an empirical cycle).

As a first step towards addressing the problem statement, we looked into how TD remediation occurs in practice and its evolution across time. To this end, we formulated RQ_1 : *How is TD paid back over the course of time and what is the difference between Python and Java?* Specifically, we looked into the TD fixing rate across projects

Figure 1.1: Problem decomposition overview

and across issue types, the repayment effort and the survival time of TD items. Our study extended a previous study on the same topic that used several Java projects. In contrast, we analyzed Python projects in order to extend the external validity of the results and be able to compare between the two languages. This is especially important since Python code is more change-prone due to having a higher number of dynamic features. Obtaining an overview of technical debt repayment in Python projects can also be useful for guiding the adaptation of TD management practices for this language, esp. in comparison with other languages such as Java. Analyzing TD remediation can help researchers gain a better understanding of the overview of TD evolution and obtain a preliminary knowledge of how developers manage TD. The study required the mining of over 60K commits, detailed code analysis on 3.7K system versions, and the analysis of almost 43K fixed issues. In addition, we relied on SonarQube to identify technical debt through the history of the analyzed projects to conduct this study (and the following studies in this thesis). SonarQube is one of the most widely used tools for assessing the level of technical debt present in a software system (Digkas et al., 2020; Amanatidis et al., 2020). According to a recent overview of TD tools (Avgeriou et al., 2020), SonarQube is the most popular tool based on its usage in the scientific literature and online media channels.

During the investigation of RQ₁, we found that some TD types frequently co-occur, which poses a great risk as it can intensify the negative consequences caused by individual TD items (Palomba et al., 2018b; Yamashita and Moonen, 2013). Although co-occurring TD has been previously investigated, there is little work beyond the co-occurrence of code smells. In addition, none of the current research examines the time interval between co-occurring issues appearance and disappearance; the survival time of co-occurring issues can provide insights on how they are repaid in practice. This lead to the next knowledge question (RQ₂): *What TD types are involved in co-occurring debt items and how long do they survive?* Consequently, we conducted a case study to investigate which types of TD frequently co-occur and how long is the interval between their appearance and disappearance. The study focused on analyzing the co-occurring associations between five types of TD in 3862 files of 20 Python projects from the Apache Software Foundation.

Another observation that stemmed from answering RQ₁ was that a considerable portion of the TD items were fixed by the same developers who introduced them; we call this *self-fixed technical debt*. Investigating self-fixed technical debt can shed light on how and why developers prioritize the remediation of different types of TD, either their own or of others. Hence, the next knowledge question (RQ₃: *How does self-fixing influence the prioritization of TD repayment?*) focuses on the phenomenon of self-fixed technical debt. As this is a relatively new concept, its investigation is twofold. The first step attempts to answer the question (RQ_{3,a}): *What TD types*

are commonly self-fixed and how long does self-fixed TD survive? This may help to understand why certain types of TD are self-fixed while others are not; it may also highlight cases where TD lives long in systems before it gets fixed, thus forcing developers to pay high amounts of interest. In particular, we analyzed how much TD is self-fixed, which types of TD are more likely to be self-fixed and how long self-fixed TD survives. The study comprised the analysis of five types of TD in more than 17,000 commits from 20 Python projects.

The answer to RQ_{3,a} provides valuable insights, but only by analysing source code, not based on the opinion of software developers. Consequently, there is no evidence of whether developers consider the differences between various debt types when repaying their own debt and whether they take other factors into account. Thus, the second step (RQ_{3,b}: *Do practitioners intentionally self-fix TD and why?*) is to investigate the phenomenon of self-fixed TD from the perspective of practitioners. To this end, we conducted a survey with 17 well-known Java and Python open-source software communities to consider the practitioners' intent and experiences. The survey included questions on the practitioners' attitude towards introducing and repaying TD in general, and regarding five types of debt: Code Debt, Defect Debt, Design Debt, Documentation Debt and Test Debt. The survey also looked into practitioners' reasons to introduce and self-fix TD and human factors that may influence their repayment actions (e.g., experience and contribution levels).

Having investigated TD remediation both from source code analysis and from the perspective of developers in the previous research questions, we realized that we are only looking at the end result, instead of the process that ended up with that result. In other words, we are missing the full picture of TD repayment: from the time a TD item is incurred in a commit until it is fixed with another commit. This wider perspective cannot be thoroughly examined by looking at a single data source (e.g., source code or developers conversations). Thus, the final knowledge question (RQ₄: *What is the process between incurring a TD item and paying it back?*) looks at two data sources (issue trackers and source code) to investigate the timeline of TD items incurred in source code, reported and discussed in issue trackers and then paid back in source code. In this study, we manually extracted 3,000 issues and analyzed TD information in almost 15K pieces of relevant issue documentation from five open-source projects.

1.4.4 Empirical Research Methodology

The previous section decomposed the problem statement into a set of knowledge questions. The answer to each knowledge question follows one or more empirical cycles, and these cycles correspond to the empirical studies conducted during this

PhD project. The empirical research was designed based on the practice of evidence-based software engineering (EBSE), which is a paradigm advocated by Kitchenham et al. (2004) in their pioneering work.

Table 1.1 presents the research methods used in each empirical study to answer the stated knowledge questions. This table also provides a reference to the sections in this dissertation that report on the corresponding empirical study design. Next, we briefly describe these empirical methods and the background of their application in this PhD project.

Table 1.1: Overview of research methodology

Code	Knowledge Question	Empirical method	Described in
RQ ₁	How is TD paid back over the course of time and what is the difference between Python and Java?	Case study	Section 2.2
RQ ₂	What TD types are involved in co-occurring debt items and how long do they survive?	Case study	Section 3.2
RQ _{3,a}	What TD types are commonly self-fixed and how long does self-fixed TD survive?	Case study	Section 4.3
RQ _{3,b}	Do practitioners intentionally self-fix TD and why?	Survey	Section 5.3
RQ ₄	What is the process between incurring a TD item and paying it back?	Case study	Section 6.3

Case study is an empirical research method that provides the means to understanding a particular phenomenon in its context (Runeson et al., 2012). Case studies are more suitable for examining relationships and do not primarily aim at establishing causality. This research method was broadly used in this dissertation: to study the TD remediation in Python projects; to investigate the phenomena of co-occurring and self-fixed TD; to explore the timeline of TD remediation.

Survey is a comprehensive research method for describing, comparing, or explaining knowledge and behavior (Fink, 2003). A survey is typically conducted by researchers to gather large sets of data from a specific number of respondents with regards to a research subject. This research method suits our goal of collecting and explaining practitioners' perceptions on self-fixed TD.

1.5 Overview of the Dissertation

The main body of this dissertation contains five chapters, i.e., Chapters 2-5, which have already been published in peer-reviewed conferences and journals, and Chapter 6, which is under submission in a peer-reviewed journal. The PhD student was

the first author and the main contributor to all studies, and the other authors were the supervisors. Each chapter aims at answering one of the research questions presented in Section 1.4.3 and is described in the following paragraphs. Finally, Chapter 7 concludes this dissertation, by summarizing the results obtained from all empirical studies and discussing future work opportunities. Table 1.2 shows the research questions and the chapters in which they are addressed.

Table 1.2: Overview of dissertation

Research Question	Chapter
RQ ₁ : How is TD paid back over the course of time and what is the difference between Python and Java?	Chapter 2
RQ ₂ : What TD types are involved in co-occurring debt items and how long do they survive?	Chapter 3
RQ _{3,a} : What TD types are commonly self-fixed and how long does self-fixed TD survive?	Chapter 4
RQ _{3,b} : Do practitioners intentionally self-fix TD and why?	Chapter 5
RQ ₄ : What is the process between incurring a TD item and paying it back?	Chapter 6

Chapter 2 is based on a published article in the Journal of Software: Evolution and Process (JSEP) (Tan, Feitosa, Avgeriou, and Lungu 2021). In this paper, we analyzed the evolution of technical debt in 44 Python projects belonging to the Apache Software Foundation, including the type and amount of TD that was paid back. We also compared our findings to a similar study in Java (Digkas et al., 2018) to understand how the unique characteristics of Python affect TD remediation.

Chapter 3 is based on a paper published in the 2020 Euromicro Conference on Software Engineering and Advanced Applications (SEAA) (Tan, Feitosa and Avgeriou 2020). This paper reports on an empirical study that investigates the relationship between different, co-occurring issues and types of technical debt. The study focuses on the issues and types of TD that co-occur frequently and on the interval between their introduction and removal. We considered approx. 28,200 commits from 20 Python projects from the Apache Software Foundation and investigated five types of TD: Design, Code, Defect, Documentation and Test Debt.

Chapter 4 is based on a paper published in the 2020 International Conference on Technical Debt (TechDebt) (Tan, Feitosa and Avgeriou 2020) and its extension, published in the Information Software Technology (IST) (Tan, Feitosa and Avgeriou 2021). These two papers are, to the best of our knowledge, the first studies to exclusively focus on self-fixed technical debt. In this work, we investigated the extent of the phenomenon of self-fixed technical debt, the types of TD that are more likely

to be self-fixed and the survival time of self-fixed TD, especially compared to non-self-fixed TD. We also examined the relationship between human factors related to the effort of developers and development teams against the self-fixing rate, together with exploring the potential aspects that may influence developers in fixing their own TD.

Chapter 5 is based on a paper published in the 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME) (Tan, Feitosa and Avgeriou 2021). In this paper, we conducted a survey with 17 well-known Java and Python open-source software communities to examine the phenomenon of self-fixed TD from the perspective of practitioners. In particular, we investigated the practitioners' attitude towards introducing and repaying TD in general and regarding five types of debt: Code Debt, Defect Debt, Design Debt, Documentation Debt and Test Debt. We also looked into practitioners' reasons to introduce and self-fix TD, as well as human factors that may influence their responses (e.g., experience and contribution levels).

Chapter 6 is an original work that is currently under review in a peer-reviewed journal. In this paper, we manually extracted 3,000 issues and analyzed TD information in almost 15K pieces of relevant issue documentation (i.e., issue summaries, descriptions and comments) from five open-source projects. Then, we extracted code commits together with commit messages from GitHub, and used a regular expression-based approach Fischer et al. (2003) to match each analyzed issue that contained TD to the corresponding commit. Next, we used *git blame* to trace the first commit in which developers introduced the pieces of code incurring TD.

Based on:

Jie Tan, Daniel Feitosa, Paris Avgeriou, Mircea Lungu, (2021) “*Evolution of Technical Debt Remediation in Python: A Case Study on the Apache Software Ecosystem*,” in *Journal of Software: Evolution and Process*, vol. 33, no. 4, e2319, DOI:10.1002/smr.2319

Chapter 2

Evolution of Technical Debt Remediation in Python: A Case Study on the Apache Software Ecosystem

Abstract

In recent years, the evolution of software ecosystems and the detection of technical debt received significant attention by researchers from both industry and academia. While a few studies that analyze various aspects of technical debt evolution already exist, to the best of our knowledge, there is no large-scale study that focuses on the remediation of technical debt over time in Python projects – i.e., one of the most popular programming languages at the moment. In this chapter, we analyze the evolution of technical debt in 44 Python open-source software projects belonging to the Apache Software Foundation. We focus on the type and amount of technical debt that is paid back. The study required the mining of over 60K commits, detailed code analysis on 3.7K system versions, and the analysis of almost 43K fixed issues. The findings show that most of the repayment effort goes into testing, documentation, complexity and duplication removal. Moreover, more than half of the Python technical debt in the ecosystem is short-term being repaid in less than two months. In particular, the observations that a minority of rules account for the majority of issues fixed and spent effort, suggest that addressing those kinds of debt in the future is important for research and practice.

2.1 Introduction

Technical Debt (TD) is a metaphor used to describe a trade-off between the short-term benefits of “cutting corners” in software development and the long-term sustainability of a software system (Li et al., 2015). Incurring technical debt through these shortcuts can bring benefits in terms of time and/or resources (e.g., shorter

time to market, less effort required to implement a feature). However, this expediency has a detrimental effect on the future maintainability and evolvability of the system, as it becomes increasingly harder to make changes (Kruchten et al., 2012; Digkas et al., 2018). This hardening translates into an additional effort for developers and, in turn, costs that can even become prohibitive to organizations. If not repaid, TD can lead to severe quality problems, unexpectedly cost overruns and substantial financial loss on software maintenance (Tom et al., 2013). It can even lead to crisis points when entire components or the system need to be replaced (Besker et al., 2017).

Although the technical debt community has reached a basic consensus on the concepts around TD (Avgeriou et al., 2016), we still need to understand better the realization and impact of TD in practice. One important aspect of that is how TD evolves over time and how it is paid back. Obtaining such knowledge can be useful, among others, for urging refactoring activities when the amount of TD increases fast, or for preventing the accumulation of new TD if repayment strategies are too costly.

Several studies that analyzed technical debt evolution have primarily focused on software written in Java (Digkas et al., 2018, 2017; Marinescu, 2012). Few other languages besides Java have been studied in the context of TD, including C# (Sharma et al., 2017) and C/C++ (Biaggi et al., 2018). The Tiobe Index¹ ranks Java as the most popular programming language, with C coming in second place and Python in third. At the time of writing this chapter – the beginning of 2020 – GitHub contains more than 1M open source Python projects and almost 1M involved developers².

Despite the huge importance of Python in current software engineering practice, reflected also in several recent empirical studies (Ma et al., 2017; Alexandru et al., 2018), we are not aware of any studies that focus on the evolution of technical debt in Python. This raises the question of whether and to what extent the results of the existing studies on TD evolution are applicable to Python. If we compare for example Python and Java, although they are both interpreted languages, they represent two different schools of thought: Java is statically typed while Python is served by a dynamic type system. Generally, developers have to spend extra effort on software maintenance (Wang et al., 2015) and software quality improvement (Ray et al., 2014) in Python, since Python code is more change-prone due to having a higher number of dynamic features (Wang et al., 2015). In addition, Python code containing dynamic features is inserted or updated more frequently when fixing bugs (Chen et al., 2018).

---

¹One of the best known indexes of programming languages popularity, <https://www.tiobe.com/tiobe-index/>, visited January 2020

²<https://github.com/search?l=Python&q=language%3APython&type=Repositories>

Furthermore, Python code is characterized by a unique kind of technical debt: Python has two major versions in use (i.e., Python 2 and Python 3), and developers often add workarounds and mix code from the two versions. Even though there is consensus that the latest version should be the *de facto* standard, it is common to still use Python 2 libraries since old code may rely on them. This kind of backward compatibility results in the usage of redundant and complicated features of the language. To make matters worse, as of January 1st 2020, Python 2 is no longer maintained³; this will cause security vulnerabilities.

Given these unique characteristics of Python, studying the TD evolution of Python programs can guide developers specifically in managing TD for this language. It can also highlight the differences with other languages, e.g. Java; this can be used to further help developers, team leaders or teams that transition from Java to Python projects or work in both languages.

In this study, we investigate the evolution of technical debt remediation in Python; in other words, how technical debt is paid back along time. Our scope is TD at the source code level: we use static source code analysis to examine how TD issues are fixed. Specifically, we (1) provide an overview of the fixing rates for a number of projects, and investigate whether the fixing rates of issues differ among projects (as each project has a different size and number of issues); (2) examine the prevalence of the various types of issues; (3) analyze the fixing rate of the most frequent issues; (4) investigate the remediation effort for the various issues; (5) look at the remediation time for various types of debt.

Furthermore, to understand how the aforementioned unique characteristics of Python affect TD remediation, we compare our findings to a similar study in Java (Digkas et al., 2018). We expect to observe several differences between Python and Java due to the aforementioned aspects: static vs. dynamic type system and transition from Python 2 to Python 3. In addition, we highlight technical debt that is language-dependent, and further discuss the possible reasons for the differences, both in terms of Python features and the transition from Python 2 to Python 3.

Our findings show some similarities between Python and Java. 20% to 30% of TD issues have a great chance to be fixed in large projects, while smaller projects have lower fixing rates. Furthermore, issues related to duplicated code require the most remediation effort, and the majority of issues with the highest fixing rate are language-specific (i.e., either Python-specific or Java-specific). In addition, the majority of issues are fixed relatively quickly while a minority of issues can live for a long time. Finally, issues that pertain to the same type of TD (e.g., design debt or defect debt) tend to have similar survival times.

On the other hand, we found a considerable amount of debt repayment that is

---

³<https://www.python.org/doc/sunset-python-2>

specific to Python. For example, test and documentation debt account for almost half of all the fixes and the majority of the remediation effort, while significant debt repayment is related to changes between major versions of the Python interpreter.

These findings can serve both researchers and practitioners: researchers can benefit from the knowledge in future approaches for automated and semi-automated debt remediation; practitioners can consider the highlighted findings and suggestions as a starting point for discussions about establishing their own practices and guidelines.

The remainder of this chapter is structured as follows. Section 2.2 presents our study design, elaborating on the research questions, as well as data collection and analysis. Section 2.3 reports the results of our study and Section 2.4 offers a discussion of the results. Section 2.5 discusses the threats to the validity of our study. After the discussion of related work in Section 2.6, Section 2.7 concludes the chapter.

2.2 Study Design

In this section, we describe the design of our study. We first state our goal and elaborate on the derived research questions. Then, we present and justify the population of the study (i.e., the selected cases). Next, we discuss the variables and procedures for the data collection, i.e., technical debt identification. Finally, we describe the analysis method for each question and sub-question.

2.2.1 Objective and Research Questions

In this section, we follow the Goal-Question-Metric (GQM) approach (Van Solingen et al., 2002) to describe the goal of our study. The GQM is a measurement paradigm based on three levels: conceptual, operational and quantitative. The conceptual level (i.e., the Goal) is defined with respect to the object of study, the purpose, the focus, the stakeholders and the context. The operational level (i.e., the Question) regards a set of questions to describe the assessment or achievement of the goal that is defined at the conceptual level. Finally, the quantitative level (i.e., the Metric) regards a set of metrics that can be measured to address each question of the operational level in a measurable way. The goal of our study is to “*analyze software systems written in Python for the purpose of understanding technical debt remediation with respect to its evolution and comparison with Java, from the point of view of software developers in the context of object-oriented open source software*”. This objective is further refined in terms of the following research questions:

RQ₁ How does TD remediation evolve in Python projects?

- RQ_{1.1}** How does the issue fixing rate vary for different projects?
- RQ_{1.2}** What is the fixing prevalence for various issues?
- RQ_{1.3}** How does the fixing rate vary for different issues?
- RQ_{1.4}** How is the effort of debt repayment distributed over different issues?
- RQ_{1.5}** After how much time is technical debt paid back?

RQ₂ How does the evolution of TD remediation in Python projects compare to that of Java projects?

RQ₃ How are the differences between Python and Java explained?

- RQ_{3.1}** How are Python features associated with the explanations?
- RQ_{3.2}** How is the transition from Python 2 to Python 3 associated with the explanations?

This is one of the first studies to examine TD in Python and, thus, the answers to RQ₁ have great value to both researchers and practitioners. An overview of TD remediation in practice can inform researchers about the generalization of existing findings in scientific literature. For example, it can provide evidence about to what extent remediation strategies at source code level are common among open-source projects or even among object-oriented languages; this may in turn drive the proposal of guidelines or development of tools. Furthermore, practitioners can use the results to understand the TD repayment in their own projects. For example, a development team can use the findings to discuss how much TD related to source code issues they expect to incur and how much they plan to pay back in the near future.

We note that research sub-questions RQ_{1.1}–RQ_{1.5} are the same as those posed by Digkas et al. (2018) in a study to examine TD remediation in Java projects. We used the same questions in order to be able to address RQ₂, i.e. to compare the evolution of TD remediation between Python and Java. Answering RQ₂ thus allows to discuss and highlight the differences and similarities with Java. The answer to RQ₂ is particularly relevant to researchers and practitioners that are experienced with Java development. Practitioners that work on both languages, can interpret the differences in TD remediation and better understand TD analysis output. Researchers that aim at extending or fork their tools to work on Python, can use our findings to inform their decisions and consider potential changes on how to interpret the output.

Identifying the differences and similarities between the two languages raises the question of how they can be explained. We particularly focus on differences as these can provide important lessons. The answer to RQ₃ can help researchers understand the impact of Python features on technical debt, and further consider which features could lead to more debt in Python projects. Furthermore, practitioners can use the results to improve the cost estimation of their debt, e.g., when planning to migrate

a project from Python 2 to Python 3 or re-implement a component in Python 3.

2.2.2 Cases Selection

The projects that we use for studying debt evolution, and in particular TD repayment strategies are all the Python projects of the Apache Software Foundation ecosystem⁴ (hereafter referred to as the Apache ecosystem or the ASF ecosystem). We made this choice for several reasons. (1) The Apache ecosystem contains a diverse set of Python projects of various sizes (up to 1,000 KLOC), activity (up to 10K commits to master), and domains. (2) Using the ASF ecosystem results in an implicit quality filter for the analyzed projects: every project is managed by a self-selected team of technical experts and is accepted in the foundation only after an initial incubation process⁵. (3) The ecosystem has been used before in the study of Digkas et al. for technical debt repayment in Java (Digkas et al., 2018), the use of Python projects that are part of the same organization increases the comparability of the results in the two studies.

The GitHub organization of the Apache Software Foundation contains projects in more than 30 programming languages. For the purpose of our study, we used the GitHub programming language filter and selected all the 44 Python projects without filtering them by any further criterion. The majority of the projects have a long history of commits, which ensures that the evolution of technical debt symptoms can be followed over an extensive period (Marinescu, 2012). In order to focus on the complete ecosystem, we decided to include in our study also the minority of projects with a small number of commits, and thus, a short history.

Table A.2 (shown in Appendix 7.2.3) shows that these projects present considerable variation in their characteristics, i.e., their number of commits, their size as measured by counting the source lines of code (SLOC) and the number of classes, their age (in the number of days the project has existed) and the contributors. As shown in the Table A.2, these projects cover a wide range of different sizes, domains, survival times and the number of contributors, which could strengthen the external validity (Easterbrook et al., 2008).

2.2.3 Technical Debt Identification

One important step for efficiently monitoring the evolution of TD, is the selection of a tool that will be able to measure various aspects of technical debt, as accurately as

---

⁴As not all projects are tightly integrated, opinions regarding the terminology may differ. We use the term ecosystems to maintain consistency with Java study (Digkas et al., 2018; Bavota et al., 2013)

⁵<http://incubator.apache.org/policy/process.html>

possible. In the state-of-research and -practice, one can identify several tools and approaches for measuring technical debt. To detect the evolution and remediation of technical debt in the subject systems, we rely on a third-party detection tool: SonarQube, an open-source code-quality measuring and management tool (Campbell and Papapetrou, 2013). The tool was selected for three main reasons: (1) its broad usage for estimating technical debt, both in academic research studies (Digkas et al., 2018, 2017; Marcilio et al., 2019; Lenarduzzi et al., 2019) and in industry (being used by more than 1,000 companies⁶); (2) its capacity to perform multi-version analysis and thus, track the evolution and repayment of technical debt over time; and (3) the fact that the tool is based on the SQALE method (Letouzey, 2012; Letouzey and Coq, 2010), which has been published and evaluated academically (Letouzey and Ilkiewicz, 2012; Griffith et al., 2017; Fontana et al., 2016; Dale and Izurieta, 2014). Of course it is not a perfect solution for measuring TD, as a perfect solution does not exist; we expand on the limitations of using SonarQube in the Threats to Validity Section.

This section briefly introduces the main terminology and concepts that are specific to SonarQube and are critical for understanding the remainder of the chapter. The tool analyzes source code to detect code smells, vulnerabilities, and bugs, and assesses / calculates technical debt as the time estimated to fix these issues.

SonarQube uses a set of **rules**, representing desirable code related practices which, when absent, introduce technical debt. During the analysis of a project, SonarQube creates a new **issue** every time a piece of code breaks one of the rules. The rules are based on well-known sources of documented bugs and vulnerabilities such as CERT and CWE⁷, as well as the PEP⁸ (Python Enhancements Proposal), which are standards largely adopted and enforced by the Python community.

Multiversion Analysis

One limitation of SonarQube (and all the other equivalent tools that we have investigated) is that the evolutionary analysis is not incremental (Ghezzi et al., 2012). This means that a new version (e.g., originated from a new commit) is fully parsed and analyzed, even if only a single line in a single file is changed. This imposes constraints on evolution analysis since it becomes prohibitive to analyze all the commits of a system. It is only recently that researchers have started proposing models of source code that are effective at incremental parsing and modeling of software, but no off-the-shelf tools make use of them yet (Alexandru et al., 2017).

---

⁶<https://www.sonarsource.com/customers/>

⁷<https://www.sonarsource.com/products/codeanalyzers/sonarpython.html>

⁸<https://www.python.org/dev/peps/>

Considering that the majority of the analyzed Python projects have at least 187 commits (with a maximum of 10942 commits) at the time of starting this work, one must make a tradeoff in analyzing a limited but sufficient number of revisions of every system. In this study, we choose a frequency of one week, i.e., we analyze weekly versions of each project, just as it was done in the Java study we use for our comparison. Furthermore, we only analyze the versions on the master branch of the studied systems.

Issue Classification and Effort Estimation

None of the project repositories of the 44 selected systems consists exclusively of Python code. By default, SonarQube detects issues for all the source code files found in a system, and when analyzing the subject systems it produces issues in code files written in different languages (including JavaScript, Java and XML). However, due to the focus of this study we only analyze those issues that are found in Python files.

In terms of severity of the rules, SonarQube defines four levels (in decreasing order): blocker, critical, major and minor. However, we limit the severity level to blocker, critical and major, because many of the minor issues are trivial and are not what developers normally think about when they talk about technical debt (e.g., *Lines should not end with trailing whitespaces*). Moreover, the minor issues have low impact and likelihood⁹ and could, therefore, bias the results. The same decision was made in the Java study (Digkas et al., 2018) and, consequently, we are in a better position to compare the Python results with the Java ones.

From the total set of rules for Python in SonarQube, we detected violations of 127 rules in this study. If we exclude the minor rules among these 127, we are left with 56 blocker, critical and major rules¹⁰; Table A.3 shows the types and their severity. Among them, 15 rules appear both in Python and Java (marked as 🍄 symbol).

Furthermore, we grouped the rules into five higher-level technical debt categories defined by Alves et al. (2014) and Li et al. (2015): Code Debt, Defect Debt, Design Debt, Documentation Debt and Test Debt. Both Alves et al. (2014) and Li et al. (2015) contain these five categories, yet they were derived independently, and they have both been widely used in other studies (Maldonado and Shihab, 2015; Bavota and Russo, 2016). To perform this grouping, the first and second authors classified the rules independently into the five TD categories, using the description of the rules and the definition of the categories. There were disagreements in the classification of eight rules. To assess the disagreements numerically, we estimated the inter-rater

⁹<https://docs.sonarqube.org/latest/user-guide/rules/>, visited in Jan. 2020

¹⁰Compared to 160 rules detected in the Java study

Table 2.1: Migration-related issues

ID	Description and Explanation
12	"<>" should not be used to test inequality The forms '<>' and '! =' are equivalent. But in Python 2.7.3 the '<>' form is considered obsolete.
21	The "print" statement should not be used The "print" statement was removed in Python 3. The built-in function should be used instead.
31	The "exec" statement should not be used The "exec" statement was removed in Python 3. Instead, the built-in exec() function can be used.
52	Backticks should not be used Backticks are a deprecated alias for repr(). The syntax was removed in Python 3.0.
276	Access of nonexistent member Some issues are related to references to 'socket' instead of 'SocketIO'.
281	Syntax error Some issues are related to references to invalid syntax in except handler with a comma.
360	Undefined name Some issues are related to references to refactored parts of the module urllib.
394	Undefined variable Some issues are related to references to Python 2 identifiers, e.g., 'xrange', 'unicode' and 'basestring'.
432	Mixed tabs/spaces indentation Indentation is rejected as inconsistent if tabs and spaces are mixed and a TabError is raised in Python 3.

agreement using Krippendorff's alpha (Krippendorff, 2011) ($\alpha = 0.74$).¹¹ In the conflicting cases, the first and second authors discussed with the third author until they achieved consensus.

To identify and analyze issues related to version migration, the first and second authors examined the description of all 56 rules together and identified those that can be associated with the update of the Python interpreter. We also checked the source code of the issues to further confirm if the rule is migration-related. Table 2.1 presents the ID numbers, descriptions and explanations of these rules.

For every issue it detects, SonarQube assigns an estimate of how much time is required to resolve it based on the SQALE¹² method (Letouzey, 2012). SonarQube analyzes the source code and uses remediation functions to work out remediation costs for each issue type (Letouzey and Ilkiewicz, 2012). The tool uses two strategies to estimate technical debt for different types of issues: Some types of issues are assigned a **constant fixing time** (e.g., the issue *Docstrings should be defined* is assigned five minutes); some other types of issues are assigned a custom fixing time according to their particular characteristics (e.g., *Source files should not have any duplicated blocks*).

¹¹Krippendorff's inspection of the tradeoffs between statistical techniques establishes that it is customary to require $\alpha \geq 0.80$. However, where tentative conclusions are still acceptable, $\alpha \geq 0.67$ is the lowest conceivable limit (Krippendorff, 2018).

¹²<https://blog.sonarsource.com/sqale-the-ultimate-quality-model-to-assess-technical-debt>, visited in Jan. 2020

Table 2.2: Number of fixed and open issues for priority type

	Fixed Issues	Open Issues	Fixing Rate
Blocker	15	39	27.78%
Critical	2,189	3,138	41.09%
Major	40,461	49,311	45.07%

Fixed Issue Detection

When analyzing multiple revisions of a system, SonarQube tracks the issues that are fixed and therefore, the debt that is repaid. According to the documentation of SonarQube¹³, issues flow through a lifecycle, and being assigned the “fixed” status could be due to two cases: (1) issues have been corrected (i.e., the issues are actually fixed); (2) or the file is no longer available (removed from the project or renamed). The second case is problematic for our purpose, as it is doubtful whether the developers actually aimed at fixing the technical debt by deleting or renaming a file. Thus, we filter out the issues that are marked as fixed due to the disappearance of the file name (e.g., *deletions* and *renames*) and remove them by using the SonarQube API. The same method was used in the Java study of Digkas et al. (2018).

Table 2.2 shows the number of issues in the analyzed systems once the Minor issues are filtered out. In total 54 Blocker, 5,327 Critical and 89,772 Major issues have been *actually detected* in the whole history of 44 systems under study. This aligns with the Java findings, where the Major issues had the highest percentage of fixes.

2.2.4 Data Analysis

This study detected more than 288K issues by analyzing over 3.7K weekly commits. If we exclude the minor issues, we are left with a total of 95K issues. Next, these data are analyzed to answer the research questions as follows.

To answer **RQ_{1.1}**, we calculate the issue fixing rate for each project, i.e., the percentage of issues that are fixed in each project, and discuss their distribution. Since different projects have a different number of fixed issues and lines of code, we investigate the relationship between SLOC and the number of fixed issues during the evolution as well as the issue-fixing rate.

For **RQ_{1.2}**, we sum up all the fixed issues grouped by rules without differentiating among projects and calculate the prevalence, i.e., the percentage of issues fixed for each rule from the total number of fixes. Then we sort the prevalence rates of all

¹³<https://docs.sonarqube.org/latest/user-guide/issues/>

the rules to get the most frequently fixed issues and classify them according to the aforementioned high-level technical debt categories.

In **RQ_{1.3}**, we compute the issue fixing rate for each rule to investigate whether some rules are fixed more often than others. However, since some rules only appear in a few projects, we introduce the number of projects in which violations of the rule appear as a reference value.

RQ_{1.4} aims at investigating the effort required to fix the technical debt issues. To answer this research question, we sum up all the effort required to fix the issues according to the estimates provided by SonarQube. To also verify the relationship between remediation effort and frequency of fixes, we compare the rank of the rules based on effort against the ranks from **RQ_{1.2}** and **RQ_{1.3}**.

To investigate **RQ_{1.5}**, we calculate and analyze the survival time of each issue. This variable is measured as the number of days between the introduction of an issue and the moment when it is fixed in the source code.

To answer **RQ₂**, we compare the findings of the aforementioned sub-questions with those reported in the Java study. In particular, we describe the similarities and differences regarding each sub-question.

Finally, to answer **RQ₃**, i.e., to explain the observed differences between Python and Java, we focus on the characteristics of the Python programming language (**RQ_{3.1}**) and the transition from Python 2 to Python 3 (**RQ_{3.2}**). For **RQ_{3.1}**, we mainly investigate the Python-specific TD and discuss the differences in the TD categories that are caused by Python features. For **RQ_{3.2}**, we focus on the issues that are caused by the backward incompatibility between Python 2 and Python 3 and further discuss the possible influence of updating Python versions for TD evolution.

2.3 Results

In this section, we present our findings and answer each research question. We note that **RQ₁** investigates the evolution of TD remediation in Python through five sub-questions, whereas **RQ₂** compares the results of **RQ₁** with those observed in Java by Digkas et al. (2018). Therefore, in Section 2.3.1 we address both research questions together and, for each sub-question, we first present the findings regarding Python and then compare to those from the study in Java. Finally, in Section 2.3.2, we elaborate on the results for **RQ₃**.

Figure 2.1: Percentage of fixed issues (black) in each project.

2.3.1 Evolution of Technical Debt Remediation in Python & Comparison to Java

Fixing Rate Variation among Projects

Figure 2.1 presents the issue fixing rate (dark colour), i.e., the number of closed issues divided by the total number of issues, for every Python project. Moreover, the projects are sorted in decreasing order of their fixing rate. More than half of the issues were fixed in the first six projects; however, the number of issues are quite different among those projects. For example, INCUBATOR-MXNET has the largest number of total issues (almost 30K issues), while less than 200 issues appear in INCUBATOR-SENSOFT-USERALE-PYQT5. At the opposite spectrum, the fixing rates of eight projects are less than 5%. These projects have small SLOC (less than 1.4K) and number of total issues (less than 161). The only exception is INCUBATOR-SDAP-EDGE with 4.9K SLOC and 940 issues and only has two commits during its evolution, which is also the minimum value in our dataset.

From Table A.2, we noticed that the projects have a median value of 187 for the number of commits. This seems to indicate that a certain number of projects are small. To further investigate the relationship between fixing rates and the number of commits, and to better compare with the fixing rates of Java projects, we conducted a separate analysis for projects with small and large number of commits. Figure 2.2 illustrates box plots depicting the distribution of fixing rates for small (less than 187) and large (more than 187) number of total commits for all the projects. It is obvious that projects with small number of commits tend to have more extreme values of

Figure 2.2: Distribution of fixing rate for small and large number of total commits

fixing rates and the majority of them have a fixing rate of less than 5%. However, the fixing rates of larger projects are much higher and the median value is 27.8%.

Furthermore, we analyzed the SLOC and the absolute number of fixed issues in each project. We observed a similar trend on SLOC and the number of fixed issues, i.e., **technical debt is more likely to be repaid in larger projects during their evolution**. To verify this observation, we sought to calculate the correlation coefficient between all four variables. For that, we first used the Kolmogorov-Smirnov test (Field, 2017) to check if the data of each variable fails to follow the normal distribution. The results showed that only the fixing rate is normally distributed (p-value = 0.33¹⁴). Following these results, we decided to verify the observation by calculating the Spearman correlation (Zar, 1972). The result between SLOC and the number of fixed issues showed to be positively strong¹⁵ ($\rho = 0.79$). Moreover, we found that the fixing rate has a strong positive correlation with the number of fixed issues ($\rho = 0.76$), and a medium positive correlation with the number of total issues ($\rho = 0.42$) and SLOC ($\rho = 0.33$).

Comparison. Similarly to the Java study, we observe a near-exponential decrease trend of the fixing rate (see Figure 2.1), where a small percentage of projects (about 10%-15%) from both Python and Java samples show rates above 50%. Moreover, both Python and Java samples contain only two projects that each display fixing rates more than 70%. One possible explanation for these two Python projects is that the majority of their issues are related to the transition from Python 2 to Python 3 (we elaborate further on this kind of technical debt in Section 2.3.2); whereas for the two Java projects with the high fixing rates, Digkas et al. (2018) suggest that it may be related to a systematic use of SonarQube by their developers to identify and fixes issues in maintenance activities. Furthermore, we noticed that the fixing rates for

¹⁴The Kolmogorov-Smirnov test verifies if values deviate from the normal distribution and, thus, a statically significant results means that values are not normally distributed.

¹⁵We interpret the correlation coefficient according to Cohen (1988), i.e., no correlation when $0 \leq |\rho| < 0.1$, small correlation when $0.1 \leq |\rho| < 0.3$, medium correlation when $0.3 \leq |\rho| < 0.5$ and strong correlation when $0.5 \leq |\rho| \leq 1$.

Figure 2.3: Distribution of fixed issues for the 56 rules detected in this study. Every bar represents a rule, height proportional to the number of fixed issues

larger Python projects are similar to those in larger Java projects: the median fixing rate for projects that contain more than 1000 commits, is between 20% and 30%.

Summary. In RQ_{1.1}, we asked how the issue fixing rate vary for different projects. We showed that the fixing rates for different projects vary greatly, indicating a wide variation in the repayment practices. To partially answer RQ₂, we compared these results with those from the Java study and concluded that despite dissimilar rules, the fixing rate in the studied sample of Python projects shows a similar distribution with the fixing rate observed in Java projects. Moreover, larger projects have a high probability that around 20% to 30% TD issues will be fixed during their evolution regardless of the programming language.

Fixing Prevalence among Different Kinds of Debt

The chart in Figure 2.3 presents the distribution of the number of fixed issues for each of the 56 Python rules detected in our project set by SonarQube. The figure shows a strongly skewed distribution where issues corresponding to about a dozen rules are fixed overwhelmingly more often. Calculating the Gini index, estimated to be the most appropriate single measure of inequality (Gastwirth, 1972), we obtain a very high value of 0.814, which indicates a very unequal distribution across the rules. We thus conclude that **most of the issues fixed during repayment concern only a small number of rules.**

Table 2.3: Stratified random sampling for 426 issues

Rule	Prevalence	Issues ¹	TD ²	Fixed ³
Docstrings should be defined	30.49%	132	100%	100%
Lines should not be too long	14.88%	64	100%	100%
Undefined variable	8.50%	37	100%	100%
Lines should have sufficient coverage by tests	6.26%	27	100%	81.48%
Source files should have a sufficient density of comment lines	5.19%	22	100%	81.82%
Access of nonexistent member	3.37%	15	100%	100%
The “print” statement should not be used	3.16%	14	100%	100%
Cognitive Complexity of functions should not be too high	3.12%	13	100%	100%
Functions, methods and lambdas should not have too many parameters	2.90%	13	100%	100%
Mixed tabs/spaces indentation	2.86%	12	100%	100%
Undefined name	2.66%	11	100%	100%
Statements should be on separate lines	2.29%	10	100%	100%
“\” should only be used as an escape character outside of raw strings	2.11%	9	100%	100%
Sections of code should not be “commented out”	2.03%	9	100%	100%
Function names should comply with a naming convention	1.90%	8	100%	100%
Source files should not have any duplicated blocks	1.72%	7	100%	100%
Syntax error	1.30%	6	100%	100%
Control flow statements should not be nested too deeply	1.24%	5	100%	100%
Functions should not be too complex	0.77%	3	100%	100%
Functions should not contain too many return statements	0.60%	3	100%	100%
Collapsible “if” statements should be merged	0.48%	2	100%	100%
Two branches in a conditional structure should not have exactly the same implementation	0.37%	2	100%	100%
Files should not have too many lines of code	0.19%	1	100%	100%
Nested blocks of code should not be left empty	0.16%	1	100%	100%
Total	98.54%	426		

¹ The number of sampled fixed issues

² The percentage of sampled issues that may present technical debt

³ The percentage of TD issues that were actually fixed

However, this conclusion depends on the accuracy of SonarQube to detect TD issues that are fixed. To assess SonarQube’s limitations, the first and second authors manually analyzed 1% of the number of total fixed issues (i.e., ≈ 430) to investigate whether those issues represent technical debt and whether their evolution is accurately captured. According to the results, the number of fixed issues are unequally distributed across the rules. Thus, we randomly selected issues by using *stratified random sampling* (Shull et al., 2007), which is used to estimate population parameters efficiently when sub-populations have substantial variability (Cochran, 2007). For each rule, we randomly selected a number of issues based on its fixing prevalence and checked their source code to verify whether they may represent TD according to the definitions of Alves et al. (2014) and Li et al. (2015). In addition, we checked if the issue was actually fixed.

The results of our analysis are presented in Table 2.3. We clarify that the issues of 32 rules were not checked, as their prevalence, stratified to 1%, was near zero (i.e., no issues to select). However, the fixing prevalences of the 24 considered rules account for 98.54% of all fixes in the analyzed projects. A total of 426 issues have

Table 2.4: Fixing Prevalence for Top 15 Rules

# ¹	Category	Severity	Rule	Prevalence
1	Documentation	Major	Docstrings should be defined	30.49%
2	Code	Major	Lines should not be too long	14.88%
3	Defect	Major	Undefined variable	8.50%
4	Test	Major	Lines should have sufficient coverage by tests	6.26%
5	Documentation	Major	Source files should have a sufficient density of comment lines	5.19%
6	Defect	Major	Access of nonexistent member	3.37%
7	Defect	Major	The "print" statement should not be used	3.16%
8	Design	Critical	Cognitive Complexity of functions should not be too high 🚩	3.12%
9	Code	Major	Functions, methods and lambdas should not have too many parameters	2.90%
10	Defect	Major	Mixed tabs/spaces indentation	2.86%
11	Defect	Major	Undefined name	2.66%
12	Code	Major	Statements should be on separate lines 🚩	2.29%
13	Code	Major	"\" should only be used as an escape character outside of raw strings	2.11%
14	Code	Major	Sections of code should not be "commented out" 🚩	2.03%
15	Code	Major	Function names should comply with a naming convention	1.90%

¹ Overall ranking across all the issue rules

🚩 Also among the most fixed issues in the Java study Digkas et al. (2018)

been randomly selected, which were all found to be potential TD issues (i.e., suit the used definitions). Moreover, only nine issues were not actually fixed.

To further investigate the research question, we examine the fifteen rules that account for over 90% of all fixes in the analyzed ecosystem (43K). Table 2.4 presents these 15 rules and the percentage of fixed issues for each, from the total number of fixed issues. As mentioned in Section 2.2, we grouped the rules into five higher-level technical debt categories, i.e., Code Debt, Defect Debt, Design Debt, Documentation Debt and Test Debt. In the following, we discuss the categories of the most fixed types of technical debt at the ecosystem level.

Rules #1 and #5 refer to Documentation Debt and account for more than one third (35.68%) of all fixed issues. *Docstrings should be defined* is by far the most fixed problem. These issues are caused by incomplete design specifications and insufficient comments in code. This paints a positive picture in the sense that the most prevalent repaid debt issues tend to be the code that is initially committed without comments and eventually fixed by adding comments. In addition, developers invest great effort in fundamental activities that can impact reuse and maintenance.

Code Debt is also of concern as it accounts for more than 26% of all fixes. Rules #2, #9, #12, #13, #14 and #15 refer to the issues found in the source code that can affect its maintainability. Moreover, we notice that these rules impact the complexity and readability of the source code. Among these rule violations, the majority (i.e., 57%) of them are related to long lines, while long parameter lists rank the second highest. The remaining four rules are all related to code conventions and rank at the bottom of Table 2.4.

Test Debt receives also some attention as rule #4 accounts for more than six per-

cent of all the fixed issues. Together, Code Debt and Test Debt fixes suggest that **reducing code complexity and boosting maintainability is paramount**.

Defect Debt includes known defects that, due to competing priorities and limited resources, should be fixed later, as well as unknown defects (Alves et al., 2014; Bavota and Russo, 2016). Table 2.4, shows that rules #3, #6, #7, #10 and #11 account for more than one fifth (20.56%) of all the fixed issues. Although it may seem that the amount of such debt is alarmingly high, in the majority of the cases, this debt concerns a kind of debt particular to Python, e.g., the transition from Python 2 to Python 3. We further discuss this matter in Section 2.3.2.

There is only one rule (#8 - *Cognitive Complexity of functions should not be too high*) appearing in Table 2.4, which belongs to Design Debt, and is of critical severity. This indicates that only about 3% of fixed issues are related to Design Debt.

Comparison. First, we notice that the distribution of fixed issues (Figure 2.3) is similar to that observed in the case of Java. Moreover, the Gini index value is also close, i.e., within 3% of the corresponding Java value (Digkas et al., 2018). In Table 2.4, the 🐘 symbol marks the rules that are also found in the most frequently fixed rules for Java (Digkas et al., 2018). Among the common rules, rule #8 also appears among the ten most fixed rule violations in the Java study, where they account for 5.4% of total fixed issues.

However, the similarities end there. The rules related to code convention (i.e., #12 and #14) are fixed at a relatively low frequency in Python, whereas they are the most frequently fixed for Java projects. Furthermore, in the Java study, four out of the top ten most addressed rules are related to Design Debt, accounting together for more than 21.5% of the total fixed issues. The difference might indicate that Design Debt violations are more urgent for Java developers. Finally, although rules related to testing and documentation account for approx. 42% of all the issue fixes in Python, they do not even make the top ten in Java. This finding may be justified by the different needs of the Python language, which are discussed in Section 2.3.2.

Summary. In RQ_{1,2}, we asked what is the fixing prevalence among different kinds of debt, which we answer as follows: a small number of issues are responsible for most of the fixes, i.e., the number of issue fixes is unequally distributed across different rules. To partially answer RQ₂, we compared these results with those from the Java study and concluded that issues related to testing and documentation are fixed much more often in Python than in Java; in contrast, Design Debt is fixed much more often in Java.

Table 2.5: Fixing Rate for Rules with at Least 500 Occurrences

# ¹	Category	Rule	Projects	Issues	Rate% ²
1	Code	"\" should only be used as an escape character outside of raw strings	20	1,312	68.52
2	Defect	Syntax error	31	972	56.89
3	Defect	Undefined variable	32	6,385	56.82
4	Defect	The "print" statement should not be used	31	2,396	56.34
5	Code	Sections of code should not be "commented out"	31	1,558	55.58
6	Design	Source files should not have any duplicate d blocks	31	1,346	54.38
7	Code	Functions, methods and lambdas should not have too many parameters	26	2,278	54.30
8	Code	Statements should be on separate lines	14	1,806	53.99
9	Code	Lines should not be too long	35	12,747	49.80
10	Documentation	Source files should have a sufficient density of comment lines	40	4,567	48.48
11	Code	Function names should comply with a naming convention	28	1,770	45.71
12	Design	Control flow statements should not be nested too deeply	27	1,265	41.98
13	Documentation	Docstrings should be defined	44	31,364	41.48
14	Defect	Access of nonexistent member	34	3,473	41.41
15	Design	Cognitive Complexity of functions should not be too high	38	3,250	40.89
16	Design	Functions should not be too complex	29	812	40.52
17	Defect	Undefined name	26	2,975	38.15
18	Design	Functions should not contain too many return statements	31	706	36.40
19	Test	Lines should have sufficient coverage by tests	44	7,595	35.17
20	Defect	Mixed tabs/spaces indentation	7	3,544	34.45

¹ Overall ranking across all the rules
² Fixing Rate

Fixing Rate Variation among Different Kinds of Debt

To compute the fixing rate for each rule, we determine the percentage of issues that are fixed from the total amount of issues corresponding to that rule. To focus on issues that are the most likely to be relevant for software developers, in the remaining analysis for this research question, we limit our discussion to rules with more than 500 issues in the ecosystem. This results in 20 rules that are presented in Table 2.5. For each rule, the table also presents the category of the rule (as already discussed in **RQ_{1.2}**), the number of issues, the fixing rate, and the number of projects in which violations of the rule appear.

By analyzing Table 2.5 we observe that the rule "\" should only be used as an escape character outside of raw strings, has the highest fixing rate (68.52%) and appears in 20 systems. However, the median fixing rate in these 20 systems is about 18%; this means that if we exclude few outlier systems, violations of this rule are not fixed very frequently. Except for this rule, the other five rules related to Code Debt all rank in the top eleven and have similar fixing rates, i.e., between 45% and 56%.

The fixing rate of the rules related to Defect Debt are found at both ends of the spectrum: some are at the high end, others at the low end. Rules #2, #3 and #4 have some of the highest individual fixing rates (more than 56%, and the latter two also appear in the table with the highest number of fixes overall from **RQ_{1.2}**). Rules #14, #17 and #20 have some of the lowest rates. Although rules #4 and #20 are commonly

violated by delaying the update to Python 3, the fixing rates are very high and very low respectively different. We further discuss the reasons in Section 2.3.2.

In Table 2.5, a quarter of the rules refer to Design Debt. All of these rules influence the quality of the source code, mostly by incurring unnecessary complexity (marked with bold keywords in the table). Arguably, complexity is a type of debt that is harder to address. Rule #6, i.e., *Source files should not have any duplicated blocks*, has the highest fixing rate among them. One possible reason is the perception that duplicated code severely complicates the maintenance and evolution of large software systems (Ducasse et al., 1999; Lozano and Wermelinger, 2008), even if emerging empirical evidence does not strongly support this assumption (Rahman et al., 2012). However, the other four rules are in the bottom half of the table with the lower fixing rates.

Almost all projects suffer from issues regarding *Documentation Debt* (40 projects for rule #10 and 44 projects for rule #13), and *Test Debt* (44 projects for rule #19). However, *Test Debt* has a lower fixing rate than *Documentation Debt*. In fact, Test Debt has the second lowest fixing rate: there is only one rule for Test Debt (sufficient code coverage), and this seems rather difficult to fix. A possible explanation is offered in the next research question where achieving sufficient test coverage shows to require the highest (repayment) effort. Finally, all fifteen issues with the most fixes (see Table 2.4) also appear in Table 2.5, which means that **the issues with the highest fixing rate are also fixed more frequently in the ecosystem as a whole**.

Comparison. The threshold of the number of rule violations, i.e., more than 500 issues, is the same one that was used in the Java study (Digkas et al., 2018). Similarly to Python, some of the rules related to Defect Debt (e.g., exception handling), also have the lowest fixing rate for Java. Furthermore, rules related to Design Debt have a low fixing rate in Python, while the Java study similarly reports that a significant part of the rules with the lowest fixing rate are related to design problems, e.g., complexity and duplication. However, unlike Python, none of the rule violations with the highest fixing rate in Java appears among the most frequently fixed ones.

Summary. In RQ_{1,3}, we asked how the fixing rate vary for different issues, which we answer as follows: different issues have a wide variation of fixing rates, even if they belong to the same debt type. However, the issues with the highest fixing rate are fixed more frequently in the whole ecosystem. To partially answer RQ₂, we compared these results with those from the Java study and concluded that the majority of the rules with the highest fixing rates are language-specific, either for Java or for Python and that many of the Python issues are caused by the backward incompatibility between versions 2 and 3.

Repayment Effort among Different Issues

To answer this question, we sum up all the effort required to fix the issues, according to the effort estimates provided for each issue by SonarQube. The total amount of repayment effort due to all the Blocker, Critical and Major issue fixes is estimated to 11,788 hours.

Table 2.6 shows the ten rules for which their fixed issues required the most effort for TD repayment. The **Effort** column represents the percentage of remediation effort for each rule, compared to the total effort. Summing up the estimated effort for the ten rules in the table adds up to more than 92.56% of the estimated effort spent in debt repayment in the ecosystem. Thus, **a minority of rules are responsible for the majority of repayment effort**. Besides effort, the table highlights two other measures: the **RT3** column shows the rank of the same issue in Table 2.5 (**RQ_{1.3}**) and **CT2** column shows the change in position with respect to Table 2.4 (**RQ_{1.2}**). These extra columns allow us to correlate the effort with the prevalence and fixing rate for a given rule. The most effort-intensive rule (to fix) corresponds to Testing Debt, which ranks the second lowest in terms of fixing rate (19) of all the rules with more than 500 occurrences. Also the second (code comments) and the third rule (docstrings) related to *Documentation Debt* have high effort in Table 2.6 (account for 25.32% of the remediation effort), but have relatively low fixing rates, hinting at a **possible inverse relationship between required effort and fixing rate**. When we compare the rank of remediation effort of the rules with their fixed issues count, we find that the majority of rules that required the most effort for technical debt repayment are not among those with the highest number of fixes. This indicates that developers prefer to fix issues that are easier first.

From Table 2.6, we notice that **test and documentation are the most costly activities in terms of remediation effort**. *Test Debt* (rule #1) requires the largest effort during evolution: almost half (46.05%) of all the effort is devoted to it. This issue is introduced in a file when its test coverage per line is less than a required threshold. The effort that is required to fix such an issue varies significantly, depending on how many lines of code in a file are inadequately covered by testing.

Moreover, we compared the remediation effort of the top ten rules to the rank of their fixed issues count (CT2 column) and fixing rates across all the rules (RT3 column). Among those rules, rule #5 (*Source files should not have any duplicated blocks*) ranks as the sixth highest fixing rate in Table 2.5 and has moved up 11 positions with respect to Table 2.4 showing the number of fixes. This rule has the largest remediation effort among all the rules related to Design Debt, and as mentioned before, it also has the highest fixing rate of all the complexity rules which have more than 500 occurrences.

Table 2.6: Remediation Effort Distribution: Ten issues are responsible for more than 90% of the remediation effort

# ¹	Category	Rule	RT3 ²	Effort ³	CT2 ⁴
1	Test	Lines should have sufficient coverage by tests	19	46.05%	↑ 3
2	Documentation	Source files should have a sufficient density of comment lines	10	16.11%	↑ 3
3	Documentation	Docstrings should be defined	13	9.20%	↓ 2
4	Defect	Undefined variable	3	5.13%	↓ 1
5	Design	Source files should not have any duplicated blocks [■]	6	3.98%	↑ 11
6	Design	Cognitive Complexity of functions should not be too high [■]	15	3.82%	↑ 2
7	Code	Functions, methods and lambdas should not have too many parameters	7	3.50%	↑ 2
8	Defect	Access of nonexistent member	14	2.03%	↓ 2
9	Defect	Undefined name	17	1.60%	↑ 2
10	Code	Function names should comply with a naming convention	11	1.14%	↑ 5

¹ Overall ranking across all the rules² Rank of the same issue in Table 2.5 (Fixing Rate Per Individual Rule)³ Remediation effort percentage⁴ Change relative to the ranking of the same issue in Table 2.4 (Fixing Prevalence for Rules)[■] Also found in the Java study of Digkas et al. (2018)

The only two Design Debt rules among the top ten rules with the most remediation effort are #5 and #6; each one of them only accounts for almost 4% of effort in Python. Furthermore, the percentage of remediation effort for Design Debt is almost twice as much as Code Debt. This indicates that **problems associated with Design Debt, e.g., duplication and complexity, require considerable repayment effort and pose critical concerns.**

Code Debt (rules #7 and #10) has a relatively low remediation effort, accounting for only 4.6% of the total remediation effort. Finally, the table shows that the effort required to resolve the problems associated with Defect Debt (rules #4, #8 and #9) follows a similar order to their fixing rates (RT3 column).

Comparison. Similarly to our study, Digkas et al. (2018) were also not able to identify a relationship between repayment effort and fixing rate. However, it is worth mentioning that rule #5 also has the biggest jump when we compare to the ranking of frequently fixed Java issues. Moreover, in both languages, the percentage of remediation effort for Design Debt tends to be higher than Code Debt. The results regarding Code Debt show another point of similarity, as rules of this type have low remediation efforts in both languages.

Unlike the findings in Python, the two Design Debt rules cost the highest pay back effort in the Java study, i.e., they are responsible for over 11% of total remediation effort respectively. Moreover, five Design Debt rules appear in the top ten Java rules with the highest amount of effort, accounting for 22.9% of the total remediation effort. In contrast, the effort for Design Debt rules in Python is around 7.8%. The result indicates that developers spend more effort to address Design Debt issues in Java projects.

Figure 2.4: Survival time histogram of 43K fixed issues with bins of 30 days

Furthermore, Test Debt and Documentation Debt account for almost half of all the effort in Python projects. However, none of the rules related to Test Debt and Documentation Debt appear in the corresponding list for Java projects.

Summary. In RQ_{1.4}, we asked how the repayment effort is distributed among different issues. To that end, we showed that a minority of rules are responsible for the majority of the repayment effort and that the variation in the remediation effort seems to be influenced by characteristics of the programming languages, as we observed different trends for some kinds of debt. To partially answer RQ₂, we compared these results with those from the Java study and found that Design Debt requires more remediation effort than Code Debt in both languages.

Survival Time of Issues

To answer this question, we analyzed the survival time of the 43K fixed issues found in this study. Figure 2.4 shows the survival time organized in bins of 30 days; it highlights that **from the technical debt that does get fixed, a large amount is fixed in a short time**. In particular, a third (34%, $\approx 15K/43K$) of the issues are fixed within one month (30 days). Moreover, a majority (60%, $\approx 26K/43K$) of the issues are fixed within two months (60 days), and an additional 27% (totaling 87%) are fixed within the first year; only a minority of issues (approx. 13%) last for a longer than a year. We also observe that the longest time it took to fix an issue in the current dataset was almost eight years.

Further, we focus on the survival time of the 20 rules with at least 500 occurrences reported (see Table 2.5 in RQ_{1.3}), including 15 rules with the most fixed issues (see Table 2.4 in RQ_{1.2}), which are responsible for more than 98.3% of the effort required for payback. Since different projects have been developed at different paces, some of them have lived for a long time and TD issues are removed at any time during this

Figure 2.5: Distribution of median survival time for 20 rules with the highest fixing rates (according to RQ_{1.3})

evolution. Therefore, the different paces may also affect the survival time of fixed issues in different projects; to mitigate this potential bias, we calculated the median survival time of each rule in each project.

Figure 2.5 shows box plots depicting the distribution of the survival median values in years (x-axis) for these 20 rules in different projects. The box plots are sorted according to Table 2.5, i.e., from high to low fixing rate (top to bottom), and are color-coded according to their TD category. To simplify the figure, we hide the outliers of these box plots.

We observe that the majority of rules (13 out of 20) have a median survival time of 50 days or even less: their survival time is shorter than the median survival time of all issues. Only one rule (i.e., *Statements should be on separate lines*) tends to survive around one year (median value). Moreover, eight out of the top ten rules with the largest remediation effort seem to have a shorter survival time (≈ 30 days), however, there is no obvious relationship between fixing rate and survival time of each rule.

Furthermore, rules from the same TD category tend to have a similar survivability distribution, e.g., the median survival time of Documentation Debt rules is about 30 days, while Design Debt has a median survival time of two months. Also, Test Debt has a relatively short and concentrated survival time compared to other debt types, whereas the majority of rules related to Defect Debt have a longer lifetime. However, violations to the six Code Debt rules have rather diverse median survival times. Issues related to long lines, (*Statements should be on separate lines* and *Lines should not be too long*) are usually the ones with the longest survivability.

Comparison. In both languages, the data indicates that the majority of the debt that is fixed, is fixed relatively soon after its introduction. Moreover, a minority of

issues can live in the system for a very long time (the longest lived issues for Python survive for eight years while for Java it is ten years).

However, technical debt seems to be repaid faster in Python than in Java. In Python, approx. 34% of the issues are fixed within 30 days and 87% within one year, whereas for Java the numbers are 20% and 50% respectively. The longest survival time of issues for Python (almost eight years) is less than Java (ten years). We further discuss that difference in Section 2.3.2.

Summary. In RQ_{1.5}, we asked after how much time technical debt is paid back. To that end, we showed that a majority of the debt that is fixed, is fixed relatively soon after its introduction. To partially answer RQ₂, we compared these results with those from the Java study and learned that: a) only a minority of issues can live in the system for a very long time (eight years the longest-lived issues for Python and ten years for Java), b) the rules from the same TD category have a similar survivability distribution, and c) technical debt seems to be repaid faster in Python than in Java.

2.3.2 Learning from the Differences between Python and Java

Python Features

Python is known for its dynamic typing system, making it popular for flexibility, expressiveness and succinctness, but also less maintainable and secure (Tratt, 2009). Altogether, the misuse of dynamic features could lead to coding issues that are often fixed by adding tests (Test Debt remediation) or exception handling (Defect Debt remediation) (Chen et al., 2018). Thus, remediation of Test Debt (i.e., *Lines should have sufficient coverage by tests*) and Defect Debt tend to appear much more frequently in Python.

Examining Test Debt closer, it is a widely spread phenomenon: Test Debt appears in all the 44 Python projects. The results of RQ_{1.3} (see Section 2.3.1) and RQ_{1.4} (see Section 2.3.1) reveal that although Test Debt has the second lowest fixing rate, it requires the largest effort. Moreover, it only has been fixed in one third (15 out of 44) of the Python projects during their evolution. To further investigate Test Debt, we focused on these 15 projects and found that they have larger sizes (45K SLOC in average) and higher longevity (around four years, with 3.5K commits in average). This confirms that Python code that is large and long-lived forces developers to increase test coverage over time (Zhai et al., 2019).

Furthermore, it is noticeable that issues related to Test Debt are not only being regularly fixed in projects with long survival time, they are also fixed relatively quickly. This indicates that in order to improve software maintainability, developers should be conscious of fixing issues related to Test Debt earlier in larger and

long-term maintenance projects. This confirms literature findings that Test Debt has a strong negative influence on software maintenance (Tufano et al., 2016; Bavota et al., 2012; Bavota et al., 2015) and high test coverage may increase software quality (Shamshiri et al., 2018; Horgan et al., 1994). Considering the Python features, we showed in Section 2.3.1 that projects with deeply nested control flow statements (i.e., Design Debt) and exception handling statements (i.e., Defect Debt) tend to be significantly less likely to be covered by tests (Zhai et al., 2019).

Finally, our results also showed that technical debt tends to be repaid faster in Python, even looking across the same rules. In addition, the longest survival time of issues among the Python projects (eight years) is shorter than among Java projects (ten years). One possible reason may be that the first Python project from the Apache Software Foundation was developed in 2006. Java appeared earlier than Python and, thus, the development time of Java projects is generally much longer than that of Python projects in the Apache ecosystem. Thus, long-lived projects are more likely to have issues with longer survival time. Another potential reason could be that, although Python is harder to maintain, its dynamic typing system also allows for smaller changes, which may facilitate maintenance activities.

Transition from Python 2 to Python 3

According to the results of **RQ_{1.1}**, there is a wide variation in the fixing rate for the projects in the dataset. We observe that both Java and Python projects contain two projects with very high fixing rates, but the reasons are different. In Python, the two projects that display the highest fixing rates (more than 78% of the issues were fixed during their evolution) are: INCUBATOR-SENSOFT-USERSALE-PYQT5 provides comprehensive user event tracking for web pages and INCUBATOR-SENSOFT-DISTILL is an analytics framework for handling and analyzing user data¹⁶.

In these projects, the majority of the issues (60% and 52%, respectively) are violations of the rule *Mixed tabs/spaces indentation*, and these issues are all fixed during their evolution. That rule is related to the transition from Python 2 to Python 3 since tabs and spaces for indentation are not allowed to be mixed in the latter. In the case of Java, Digkas et al. (2018) found that development teams use SonarQube to manage and maintain the two projects with the highest fixing rates in the Java study. This is not the case for Python: we checked the repository and commit messages for all the Python projects and could not find any evidence of using SonarQube to manage TD.

As shown in **RQ_{1.2}**, five rules related to Defect Debt account for more than one fifth of all the fixed issues. Among them, rules #7 and #10 occur with the update of

¹⁶<http://senssoft.incubator.apache.org/system/>

the Python interpreter (from Python 2 to Python 3). In Python 3, the “print” statement requires parentheses, and mixing the use of tabs and spaces for indentation is no longer allowed. Furthermore, some issues that are mapped in rules #3, #6, and #11 are also caused by the update of the Python interpreter. For these rules, we randomly selected 300 fixed issues for each rule and checked them manually. Regarding rule #3 (*Undefined variable*), over 70% of the fixed issues are related to references to Python 2 identifiers (e.g., ``xrange``, ``unicode`` and ``basestring``). For rules #6 (*Access of nonexistent member*) and #11 (*Undefined name*), only about 6% of the issues were related to naming or syntax affected by the different versions of the Python interpreter, e.g., references to ``socket`` instead of ``SocketIO``, and to refactored parts of the module `urllib`. Among them, the first two authors had disagreements only on the issues related to rule #3. However, there was a high level of inter-rater agreement between two authors when we calculated Krippendorff’s alpha ($\alpha = 0.82$) (Krippendorff, 2018). Altogether, we estimate that **a majority of the fixed Defect Debt ($\approx 60\%$) is related to incompatibilities between Python 2 and Python 3**. That is a plausible explanation of why the fixed issues related to Defect Debt account for a higher percentage of the total fixed issues in Python compared to Java ($\approx 15\%$). Moreover, unlike the effect of updating the Python interpreter, the majority of Defect Debt issues in Java projects are related to exception handling.

Although multiple Python-specific issues are related to postponing the ion to Python 3, the rule violations have a wide distribution: some rules have a high fixing rate while others a low fixing rate (discussed in **RQ_{1.3}**). This is partially in contrast to the Java study, where the majority of Java-specific rule violations tend to have a quite high fixing rate. To study this further, for rules #2 and #3 (*Syntax error* and *Undefined Variable*) in Table 2.5, we randomly sampled 300 fixed issues and manually inspected them. We found out that most of the issues are associated with the use of older versions of the Python interpreter (e.g., invalid syntax in `except` handler with a comma or the use of Python 2 specific identifiers such as `basestring`). As we did before, we also estimated the inter-rater agreement for rules #2 and #3 by calculating Krippendorff’s alpha (Krippendorff, 2011) and the values are 0.78 and 0.85 respectively. We thus estimate that almost half of the Defect Debt (47%) is related to postponing the update to the latest Python interpreter and has a high fixing rate (over 56%). In contrast, rule #20 is also commonly violated by delaying the update to Python 3, however, it has the lowest fixing rate in Table 2.5. Issues pertaining to this rule appear 3544 times in only seven projects. Although six projects have higher fixing rates (over 50%) and three of those have totally fixed all these issues, one project (TASHI) has a rate of 16% and contains more than 69% of all issues (2463 out of 3544). Such observations suggest that **some applications within the Apache ecosystem have still not migrated from Python 2**.

It is worth noting that Test Debt (i.e., insufficient test coverage) accounts for almost half of the remediation effort (approx. 46%) for Python. Since there is a great difference between Python 2 and Python 3, developers should make sure that the test suite is thorough before migrating to Python 3. Furthermore, migrating the tests before the rest of the code facilitates the identification of defects and contributes to a smoother transition. Moreover, migrating the test suites tend to be simpler than migrating code, thus it could also provide an idea of how easy it can be to migrate projects to Python 3 (*Porting Python 2 Code to Python 3*, 2017). Thus, issues related to insufficient test coverage tend to be more important in Python projects, especially for projects with a long history, since they are more likely to have migration problems with different Python versions.

2.4 Discussion

In the previous section, we reported on the evolution of TD remediation in Python and pointed out the similarities and differences compared to Java. Although the sets of SonarQube rules for the two languages do not completely overlap, we identified a number of similarities. In both languages, projects are likely to have around 20% to 30% of their TD issues fixed, with a small number of issues accounting for most of the fixes. Moreover, the majority of the rules with the highest fixing rates are related to language-specific rules. In addition, a minority of rules account for the majority of the remediation effort and Design Debt requires more effort than Code Debt. Furthermore, for the technical debt issues that do get repaid, the majority is fixed relatively soon after they are introduced. However, a minority of the issues that get fixed, stay in the system for a very long time in both languages (until they are eventually fixed).

We also found some differences, i.e., results that are unique to either Python or Java: the potential reasons for the high fixing rates tend to be different between Python and Java projects; Test Debt and Documentation Debt are fixed much more often and also require more effort in Python, while Design Debt is fixed much more often in Java; the high fixing rates for some Python rules are caused by the changes between major versions of the Python interpreter. Furthermore, we presented possible explanations for the observed differences based on particular features of the Python language and the migration from Python 2 to Python 3. This discussion was at the level of TD categories (e.g., Code Debt, Design Debt). However, the differences in the number of rules among the categories may influence the comparison, e.g., Test Debt only has one rule in the Python results.

One possible way to compare at the level of rules is to look at the rules that are

common between the two languages. To that end, we analyzed the 15 rules that are common to both Python and Java. Thus, in Section 2.4.1, we focus on these 15 common rules, revisit the research questions **RQ_{1.2}**, **RQ_{1.3}** and **RQ_{1.4}**, and discuss them. We do not revisit **RQ_{1.5}** as we could not derive the related information from the Java study. Finally, in Section 2.4.2 we focus on the implications of this work for both developers and researchers.

2.4.1 Common Rules in Both Languages

Table 2.7 shows the comparison between the findings for Python and Java for the 15 common rules, regarding the prevalence of fixed issues (**RQ_{1.2}**), the fixing rate (**RQ_{1.3}**), the effort required to fix the issues (**RQ_{1.4}**) and the percentage of projects in which violations of the rule appear. Fixing rates of the common rules have been calculated in **RQ_{1.3}**, thus, we used those values here. However, for the comparison in **RQ_{1.2}** and **RQ_{1.4}**, the results we had concerned the fixing prevalence and remediation effort for all the rules; thus, we calculated the distribution of the number of fixed issues and remediation effort among the 15 common rules. Moreover, the ‘**Aggregated**’ row on Table 2.7 shows the results from combining the 15 rules, e.g., those 15 common rules account for 11.89% of all fixed issues in Python, and 34.53% in Java.

For **RQ_{1.2}**, it is interesting that the five most prevalent fixed issues are the same in both Python and Java: the top five rules in Table 2.7. Moreover, the number of issue fixes related to these five rules account for almost 90% of common issue fixes in both Python (87.35%) and Java (89.18%). Considering all of the most prevalent issues in both languages, we note that those five rules appear in the top ten Java rules with the most fixing prevalence, and three of them appear in Python. Furthermore, among the common rules, issues related to *Cognitive Complexity of functions should not be too high* appear in the largest number of Java and Python projects (esp. in the case of Java this rule is found in all projects as shown in the last column of Table 2.7). This highlights that complexity is a major source of concern in both languages.

Regarding **RQ_{1.3}**, there is a wide variation of fixing rates in Table 2.7. The rule *Variables should not be self-assigned* seems to be a special case with a fixing rate of 100% in Java projects. However, there are only 30 issues related to that rule, so it appears to be an outlier. Excluding that extreme value, fixing rates of the rest of the common rules have similar variation intervals in Python and Java, i.e., from 19% to 56%.

For **RQ_{1.4}**, we note that the three most costly activities (i.e., rules #1, #2 and #5) are the same in both languages, and even appear in the same order. These three rules require the majority of remediation effort, i.e., more than 80% of the total effort

Table 2.7: Comparison of 15 common rules for Python and Java

# ¹	C ²	Rule	RQ _{1.2} : P% ³		RQ _{1.3} : F% ⁴		RQ _{1.4} : R% ⁵		Projects% ⁶	
			Python	Java	Python	Java	Python	Java	Python	Java
1	S	Cognitive Complexity of functions should not be too high	26.19	15.76	40.89	42.34	36.46	32.30	86.36	100
2	C	Statements should be on separate lines	19.21	22.77	53.99	40.59	1.32	1.96	31.82	91.23
3	C	Sections of code should not be “commented out”	17.06	22.56	55.58	41.44	5.85	9.69	70.45	100
4	S	Source files should not have any duplicated blocks	14.42	11.05	54.38	19.47	38.01	34.00	70.45	100
5	S	Control flow statements should not be nested too deeply	10.46	17.04	41.98	31.01	7.17	14.63	61.36	100
6	S	Collapsible “if” statements should be merged	4.04	2.56	41.75	36.69	1.38	1.10	68.18	100
7	S	Two branches in a conditional structure should not have exactly the same implementation	3.07	0.27	48.75	48.00	2.11	0.23	47.73	70.18
8	S	Files should not have too many lines of code	1.64	0.61	31.44	25.53	6.73	3.14	36.36	98.25
9	C	Nested blocks of code should not be left empty	1.32	4.50	37.22	38.24	0.45	1.93	36.36	96.49
10	C	Methods and field names should not differ only by capitalization	0.30	0.89	34.88	40.42	0.20	0.76	20.45	87.72
11	C	Redundant pairs of parentheses should be removed	0.26	0.21	44.83	32.49	0.02	0.02	15.91	56.14
12	C	A field should not duplicate the name of its containing class	0.16	0.20	28.57	23.19	0.11	0.17	25.00	89.47
13	D	Track uses of “FIXME” tags	0.91	1.16	26.59	44.92	0.00	0.00	22.72	68.42
14	C	Identical expressions should not be used on both sides of a binary operator	0.12	0.28	19.35	55.51	0.02	0.05	20.45	56.14
15	F	Variables should not be self-assigned	0.85	0.02	55.13	100	0.17	0.00	22.72	7.02
Aggregated			11.89	34.53			10.47	34.64		

¹ Ranking across the common rules
² Category (C =Code Debt, S = Design Debt, D =Documentation Debt, F =Defect Debt)
³ Prevalence percentage
⁴ Fixing rate
⁵ Remediation effort
⁶ The percentage of projects in which the rule appears

estimates for the common rules. Among them, the violations of the rule *Source files should not have any duplicated blocks* require the most estimated effort in both Java and Python, accounting for 34% and 38% respectively of the total remediation effort. Furthermore, *Cognitive Complexity of functions should not be too high* accounts for the second largest amount of repayment effort (about 32% and 36% respectively) for projects of both languages. In addition, the fixing rates for this rule are also very similar (approx. 40%).

The high fixing rates and remediation effort of common issues related to complexity (i.e., rules #1-#5) indicate that those issues are given priority despite requiring significant effort to fix; this is likely because developers understand their impact on software maintenance and try to fix them to avoid paying high technical debt interest. Moreover, three of these rules (i.e., rules #1, #4 and #5) are related to De-

sign Debt. In fact, except for two rules (#2 and #3) that belong to Code Debt, six of the top eight rules with the highest percentage of fixed issues are related to Design Debt. The majority of them also have the highest percentage of remediation effort and appear in most projects. Again, we conjecture that issues related to Design Debt are more commonly addressed by software developers because they tend to cause a lot of extra maintenance effort; despite their high remediation effort, fixing them saves effort in the long term. Consequently, we advise practitioners to prioritize the remediation of these Design Debt issues in both Python and Java. The longer these issues stay in the system, the higher the technical debt interest to be paid.

In addition, almost all of the common rules (i.e., 13 out of 15) are related to Design Debt and Code Debt, which indicates that developers might concentrate on similar issues related to code comprehension, maintenance and complexity regardless of the programming language. Although half of the rules in the Python study (28 out of 56) belong to Defect Debt, a large number of them constitute Python-specific technical debt; the two studies have only one Defect Debt rule in common. This indicates that, among all technical debt types, Code Debt and Design Debt are more independent of programming languages and have a deeper impact. We see this as an important point for technical debt tool vendors that offer support for multiple languages: Code Debt and Design Debt rules should be assigned higher weights and the commonalities in these rules across languages should be exploited. Furthermore, development teams that work with two or more languages, can have common thresholds or quality gates for Design and Code Debt issues across all languages.

Furthermore, issues with a similar remediation effort might be treated very differently, depending on the programming language. For example, although the remediation effort for the rule violation *Source files should not have any duplicated blocks* is quite similar for both languages, the fixing rates of that rule in Python and Java vary greatly, approx. 54% and 19%, respectively. Moreover, the average number of issues of this type per file is similar in both languages (approx. 0.24). We investigated this rule further and found that, for the Python projects, the average effort per issue is doubled when compared against the Java projects. SonarQube estimates the effort to fix this issue based on the size of the duplicated block, which means that duplicated blocks in Python are likely to be approx. twice the size of duplicated blocks in Java and, therefore, may receive more attention from developers.

Another interesting rule is the one with the second most remediation effort, i.e., *Cognitive Complexity of functions should not be too high*. We analyzed the issues of this rule further and noticed that despite the similarities in repayment effort and fixing rates of that rule in two languages: a) Python has considerably more issues per files than Java (approx. 0.59 and 0.24, respectively); whereas b) Java has marginally more

complex methods (based on the repayment effort per issue). This sheds light on the programming style associated with the two languages. For example, methods in Java classes can be inherently more complex, while Python offers a variety of idioms (e.g., list comprehension) that alleviates cognitive complexity. Moreover, Python modules are often implemented in a more imperative style (as opposed to object-oriented), which may, for example, result in fewer accessor methods.

2.4.2 Implications

The previous sections elaborated on both generalized and language-dependent results. Since both of them have implications for developers and researchers, we discuss these implications first for language-dependent and subsequently for generalized findings.

Lessons Learnt from the Differences

Based on the findings that are different between Python and Java projects, we highlight that most of the rules with the highest fixing rates and highest spent remediation effort are language-dependent. This limits the external validity of empirical studies (like this one) to the language studied. Researchers should consider this threat to validity and if possible, replicate studies among languages and compare results. In fact, finding differences among languages can guide the calibration of technical debt tools towards the individual features of each language, instead of simply having different rules for different languages.

Test Debt and Documentation Debt are fixed much more often and repaid very fast in Python projects. Practitioners can consider this information to prioritize their maintenance activities: they can check whether these issues are also fixed quickly in their project; if they are not, practitioners can reflect whether they have a good reason not to fix them. Moreover, researchers can focus on improving the detection of issues related to other debt types, e.g., Defect Debt and Code Debt, and try to understand why these are not prioritized by practitioners.

Although Test Debt is highly spread in Python projects, it is also likely to be resolved within the first year. This information can be used as an early warning of long-lived debt, as any Test Debt surviving for longer than a year will potentially survive for a much longer period.

A large proportion of debt (esp. Defect Debt) is associated with the migration between Python 2 and Python 3. This should urge researchers to pay close attention to debt emerging from programming language updates (e.g., from changes in syntax or added idioms) and improve related TD detectors and analyzers in time. For example, these tools can warn developers of such migration issues and promote

improved best practices early enough to mitigate further risky debt such as from the discontinuation of an entire major version.

Although the dynamic features of Python may be associated with greater maintenance efforts, they can also lead to a more cautious development style, e.g., with a stronger debt repayment culture. On the one hand, this indicates a potential maturity of the Python community in adopting the observed debt repayment practices. On the other hand, it should draw the attention of researchers to further investigate the tight relationship between the features (and associated culture) of a programming language and TD repayment practices.

Lessons Learnt from the Similarities

Based on the general findings that are the same or similar for both languages, we highlight that fixing rates for the majority of projects are not high, but that is not necessarily alarming; projects may thrive despite the presence of technical debt. Practitioners should look more at the trends (is the debt increasing or decreasing in the long term?) rather than absolute numbers (not all debt has to be repaid) and only set realistic goals for repayment, e.g., targeting issues that incur high technical debt interest.

Issues that are easier to fix also have a higher fixing rate—developers seem to be dealing with the ‘low-hanging fruits’. This unfortunately corresponds to a minority of all rules. Researchers should focus on tool support for the more complex types of debt (with low fixing rates), as they tend to survive much longer incurring developers to pay significant amounts of interest. Any approach that can support developers in fixing these ‘expensive’ issues more cost-effectively would be of added value.

Since issue fixes related to complexity account for almost 90% of common issues and cost around 90% of the remediation effort, developers should pay more attention to fixing them by giving them high priority among all issues. In addition, TD management tools should emphasize these issues and to some extent prompt developers to deal with them with high priority.

There seems to be an inverse relation between repayment effort and fixing rates, but that was not clearly established. Researchers could look into confirming this relation, or even better establishing causality between the two concepts. This would have wide implications for technical debt tools, as they would need to strongly encourage the remediation of debt with high repayment effort. Development organizations can also take this into account by encouraging debt repayment of costly issues as part of the organizational culture.

The majority of issues that do get fixed, are fixed rather quickly after being in-

roduced. In contrast, a minority of issues can live for a long time. This has two important implications. First, practitioners can bear this in mind: if they postpone fixing an issue, it will likely survive long enough to incur significant technical debt interest. Second, researchers should explore the reasons behind the long survival of certain issues. Do developers assign them a low priority because they do not consider them important enough or because the remediation effort is too high? Furthermore, researchers can build on this information to calibrate the priority weights and early warnings for such issues in tools. For example, issues that are not incurring high technical debt interest may be assigned lower priorities.

2.5 Threats to Validity

We discuss the threats to construct and external validity, reliability, and confounding factors. We note that internal validity is not relevant to our study, since we did not seek to establish causal relations.

2.5.1 Construct Validity

This type of threat pertains to the connection between the research questions and the objects of study (i.e., do we measure what we intend to measure?). In this regard, the result of this study relies on SonarQube to detect TD issues that are fixed and the amount of TD that is paid back. Although the tool is widespread in both industry and academia, our interpretation of TD is limited to the tool’s capabilities. Different strategies could be used in different tools to detect TD, which might lead to other possible definitions of technical debt rules and, consequently, potential TD issues that are fixed during the evolution. To assess SonarQube’s limitations, the first and second authors randomly selected 426 issues (i.e., 1% of the number of total fixed issues) by using *stratified random sampling*. Then, we manually checked whether those issues represent technical debt and whether their evolution is accurately captured. The results show that all of them represent technical debt and almost 98% of them were actually fixed.

Furthermore, when we analyze the evolution of projects, we choose the change history on a weekly basis and, thus, the introduction and removal time might have a maximum error of one week. Therefore, this could lead to a range of errors in calculating the survival time of fixed issues. Moreover, projects may be developed at different paces, which may also affect the survival time of fixed issues in different projects. Overall, these two threats only affect the result of RQ_{1.5}, i.e., the survival time of different fixed issues. To mitigate these threats, at least to some extent, we compare the median survival time of the issues between different rules.

2.5.2 External Validity

This type of threat concerns the generalizability of our findings. Although we analyzed all the Python systems in the Apache ecosystem, which represent a considerable corpus of evolving Python systems, we can not claim that these results can fully represent the entire population of non-trivial Python projects.

Furthermore, the set of rules considered in this study is not exhaustive and does not portray the complete set of TD related issues that may affect Python source code. Some of them are language-specific and some might even have different implementations for different versions of the tool itself. From this point of view, the comparison with Java is limited. To partially address this, we investigate the results of common rules that appear in both Python and Java projects in Section 2.4.1 independently.

Since the version of SonarQube used for this study (SonarQube 7.0) is newer than the one used in the Java study (SonarQube 6.4), we compared the sets of rules to better understand the impact of different versions of the tool in analyzing TD evolution. As a result, we only found two new common rules in the last version, i.e., *Lines should have sufficient coverage by tests* and *Identical expressions should not be used on both sides of a binary operator*. The former has a great impact on our result as we discussed in Section 2.3.2. However, the latter has a negligible effect since it only has six fixed issues, accounting for 0.01% of the total fixed issues that have been detected in our study. Finally, we note that many of the technical debt issues are language independent (e.g., method complexity and code duplication).

2.5.3 Reliability

To address reliability threats, at least three researchers were involved in both data collection and analysis. Moreover, samples of analysis output at the different steps were manually inspected for irregularities and alignment with the proposed study design. Finally, most steps were automated by scripts which are publicly available together with the collected dataset.

Moreover, the mapping of the 56 SonarQube rules into the five TD categories was performed independently by two researchers, with a third researcher helping to resolve conflicts (see Section 2.3.2). However, it is possible that different researchers might map these rules to different categories.

2.5.4 Confounding Factors

Confounding factors are variables that may affect the dependent variables without the knowledge of the researchers. In our study, the main limitations that we expect

in this regard pertain to the differences between Java and Python. Despite our attention to details while replicating the study of Digkas et al. (2018), some aspects that could not be controlled may affect the comparability of the results.

The most prominent factors are related to the characteristics of the projects. Although all (Java and Python) projects are part of the Apache foundation, the domain of the projects is not uniform among the languages. For example, NUTCH is a Web crawler relying on Apache Hadoop data structures, which has the highest fixing rate among Java projects. However, there is no Web crawler in the selected Python projects. Furthermore, although language features can affect the complexity of the project, and thus comprise valid comparisons, the complexity of the source code can also be influenced by the type of project and developers' experience, which may also not be uniform among Java and Python projects.

2.6 Related Work

The design and results of our work are more closely comparable to two studies by Digkas et al. (2017, 2018). The first study (Digkas et al., 2017) examines the number of TD issues in 66 Java projects of the Apache Software Foundation over a period of 5 years, and also use SonarQube to investigate how TD evolved and what types of issues are involved. The results show that on the one hand, there is a significant increase trend on size, number of issues and on the complexity metrics of the project, while on the other hand, the normalized TD decreases as project evolves. The follow-up study (Digkas et al., 2018) investigates the amount of TD that is paid back and the issues that are fixed (Digkas et al., 2018). An in-depth comparison with this work is presented in the results of RQ₂ and further discussed in Section 2.4.1.

Although we used SonarQube like Digkas et al. (2017, 2018), there are other means to detect TD that have been used in other studies. For example, Marinescu (2012) proposed and evaluated a framework to detect TD via metrics-based detection rules for seven object-oriented design flaws. Their case study shows how the framework can detect debt symptoms and past refactoring actions. However, they only focused on Design Debt, and their findings are based on two Eclipse projects. Unlike them, we investigate five debt types (including Design Debt) in 44 Apache Python projects. Aligning to our results, the authors also noticed that issues related to code complexity have a significant impact on the software maintenance.

More recently, Bavota and Russo (2016) detected technical debt by analyzing code comments for self-admitted technical debt instead of analyzing the source code itself. In this context, they found that self-admitted technical debt is diffused in the mined open-source projects. Although our study analyzed TD in source code, we

also found that TD issues are diffused during the evolution of Python projects.

We detected 56 different TD issues in our study, which belong to five TD categories. However, to the best of our knowledge, most of the previous studies involved only a few types of TD. For example, Olbrich et al. (2009) analyzed historical data over several years of two projects to investigate two code smells (i.e., Code Debt), namely God Class and Shotgun Surgery. The results show that they can identify different phases in the evolution of code smells during system development. While we considered 14 rules related to Code Debt in our study, these two code smells are not included. However, similar to our results, their findings also indicate files affected by Code Debt (in their study, by smells) as noticeable maintenance challenges.

Another large scale empirical study was presented by Palomba et al. (2018b), which focused on investigating the diffuseness of 13 code smells and their impact on maintenance properties. Their findings show that the most diffused smells are related to size and complexity. This observation is in line with our finding that the majority of fixed issues and effort estimates pertain to issues related to complexity. However, unlike them, we detected additional TD categories other than Code Debt, and focused on the remediation.

Our investigation of Python was partially motivated by the goal of investigating another prominent programming language besides the highly analyzed Java. A handful of other studies also aimed to do the same. For example, Sharma et al. (2017) mined 19 design smells and 11 implementation smells in 1988 C# repositories to investigate fundamental characteristics of code smells. Some of those smells were also found in our study, e.g., *duplicate code* and *long method*. They found that open-source C# programs have a high average smell density and that some smells occur more frequently. Comparing to our results, we found that TD issues are also widespread in Python projects, as well as that some issues are more prevalent.

2.7 Conclusion

In this chapter, we presented the results of a case study of the multi-year evolution of technical debt remediation in 44 Python projects from the Apache Ecosystem, focusing on the effort required for technical debt repayment and the issues that are fixed. The analysis we performed shows that most of the repayment effort goes into improving testing, adding documentation, reducing complexity, and removing duplication. We also made a comparison between the results of this study and the one on Java reported by Digkas et al. (2018), presenting similarities and differences between the two languages.

Several of the findings are aligned with the study on the Java part of the ecosystem. First, a minority of rules account for the majority of issues fixed and the spent effort, and this suggests that addressing those kinds of debt in the future might be important for research and practice. In addition, almost 90% of fixed issues and 80% of effort estimated to be invested in fixing pertain to rules related to Design Debt, e.g., complexity, those issues are of common concern to all software developers and maintainers. Furthermore, the majority of the rules with the highest fixing rate (up to 60%) are language-specific issues. Finally, a minority of issues can live for a long time and the issues from the same TD category have a similar survival time.

Some aspects of TD repayment in the Apache ecosystem are particular to Python projects: a) more than half of the Python technical debt that gets fixed is short-term, i.e., being repaid in less than two months, which is much faster than the previous study on Java has indicated; b) documentation-related debt seems more prominent in Python than in Java; and c) despite Test Debt appearing in all projects, it has only been fixed in one third of Python projects and those have a long history. Despite these indications, we also highlight that biases such as project-dependent variables (complexity, domain, etc.) may affect the comparison between program languages.

The results of our study have several implications for practitioners, particularly developers. First, our study highlights the importance of strategically managing debt rather than repaying it all. We have shown that in one of the most successful collections of open source software projects, not all debt is repaid and there seems to be a preference in the way some types are prioritized over others. Indeed, TD issues related to code complexity seem to receive high fixing priority in the Apache ecosystem. Furthermore, test coverage may require more attention during software maintenance since it tends to require more effort to be fixed; however, based on our observations, developers seem more conscious of testing issues in long-lived projects.

Finally, our results indicate that the majority of the issues are fixed in a relatively short time. Therefore, we believe that quality-minded developers reading this study will verify whether the issues in their Python projects are also as quickly addressed; noticeable delays in this regard may indicate the accumulation of substantial TD.

The findings reported in this chapter shed light on several aspects of technical debt remediation in Python projects. During the investigation of technical debt evolution, we found that some TD types frequently co-occur. To further investigate this phenomenon, the next chapter reports on the types and the survival time of the co-occurring technical debt in Python projects.

Based on:

Jie Tan, Daniel Feitosa, Paris Avgeriou, (2020) *“Investigating the Relationship between Co-occurring Technical Debt in Python,”* in Proceedings of the 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA '20), Portorož, Slovenia, pp. 487-494, DOI:10.1109/SEAA51224.2020.00082

Chapter 3

Investigating the Relationship between Co-occurring Technical Debt in Python

Abstract

Technical debt (TD) reflects issues that may negatively affect software maintenance and evolution. There is currently little evidence on how the different types of TD co-occur; for example, how code smells and design smells affect the same part of the system. This chapter investigates how different types of TD co-occur, as well as the time period of the co-occurrence. To that end, we analyzed the co-occurring associations between five types of TD, captured in 42 SonarQube rules, in 3862 files of 20 Python projects from the Apache Software Foundation. We found that this phenomenon is dominant, affecting more than 90% of Python files. We also found that Documentation Debt and Test Debt appear in the majority of the files, although it seems to be mostly by coincidence. Finally, we noticed that co-occurrence of TD seems to happen very quickly: co-occurring issues tend to be introduced within the same week. But once it does happen, it is hard to get rid of. These results can benefit both researchers and practitioners by: aiding the prioritization of TD remediation; leading to novel tools for detecting co-occurring TD and warning potential issues; shedding further light on the explanation of how TD is introduced and can be mitigated.

3.1 Introduction

Technical debt (TD) reflects shortcomings in a software system that can potentially harm its maintainability and evolvability (Li et al., 2015). Different types of TD are related to different artifacts of the software development life cycle. The majority of work on TD is related to the source code, but there is also work associated with other types of debt, related to documentation, design, defects and testing (Alves et al., 2014; Bavota et al., 2015).

The vast majority of research on TD has focused on investigating the effect of *individual* instances of TD, e.g., code smells (Sjøberg et al., 2013). More recently, researchers have been looking into the interactions among different TD instances, when they *co-occur* in the same component, class, or file. For example, one study found six pairs of co-occurring code smells that frequently appear together in open-source software systems and elaborated on the extent of this phenomenon as well as how such co-occurrences are introduced and removed by developers (Palomba et al., 2018a). Other studies confirmed that co-occurring TD poses a great risk as it can intensify the negative consequences caused by individual issues (Palomba et al., 2018b; Yamashita and Moonen, 2013).

Although the related work has investigated the negative impact of co-occurring TD instances on maintenance, they have two main limitations. First, there is very little work beyond code smells, on how other types of TD co-occur (e.g., Design Debt or Documentation Debt). A broader understanding of the different types of debt in a project is essential to better inform the involved stakeholders on various maintenance liabilities, especially because the different types of debt are related. For example, one may decide to document the code at a later moment (i.e., Documentation Debt) and this decision may lead to further creation of unnecessarily complex components (i.e., Design Debt) due to poor insight into their implementation.

Second, current research does not deal with the time interval between a co-occurrence appearing and disappearing. This may hint at possible reasons for TD co-occurrences and provide information for prioritizing their remediation. For example, if one type of debt is always followed by the appearance of a second type very soon after the first, the first may be causing the second. That could, at the very least, result in monitoring the first type from an early stage. Conversely, co-occurring issues that seem to be removed almost at the same time, can be targeted together for more effective maintenance.

This chapter reports on an empirical study that investigates the relationship between different, co-occurring issues and types of technical debt. The study focuses on the issues and types of TD that co-occur frequently and on the interval between their introduction and removal. We consider approx. 28,200 commits from 20 Python projects from the Apache Software Foundation (ASF) and investigate five types of TD: Design, Code, Defect, Documentation and Test Debt. To detect TD, we use SonarQube, an open-source tool that measures quality and technical debt. We selected this tool for two main reasons: (1) it is being widely used in industry¹ as well as in the literature of TD (Dale and Izurieta, 2014); (2) it can track the evolution of technical debt by analyzing multiple versions of projects. SonarQube detects TD by identifying violations of a number of rules; different rules correspond to different

---

¹<https://www.sonarsource.com/customers/>

types of TD. Limitations of using SonarQube are reported in Section 3.5.

We have selected projects in Python, as it has grown substantially over the years and is currently ranked as the 3rd most popular programming language². Although developers can benefit from the flexibility and conciseness brought by the dynamic features of Python, they have to spend extra effort on software maintenance and software quality improvement (Wang et al., 2015). Thus, our results can guide developers specifically in managing TD for Python and highlight the differences with other languages (Tan et al., 2020).

Our findings indicate that the phenomenon of co-occurring TD is widely spread, i.e., more than 90% of Python files have been affected by at least two types of TD at the same time. Moreover, Design Debt issues are the most likely to co-occur with issues of similar nature, while Documentation Debt and Test Debt appear in the majority of the files. Test Debt issues co-occur the most frequently with other types, although the majority of such associations does not seem strong.

We also found that co-occurring issues tend to be introduced within the same week. However, after one of the co-occurring issues is removed, the second one survives for much longer (often around a year). The most notable exception to this relates to Design Debt: co-occurring Design Debt issues are likely to be removed together within a week.

Our findings are relevant to both researchers and practitioners. For the former, the results would be useful to develop tools to prioritize TD remediation and warn about potential future co-occurring issues. For example, the co-occurring relationship between Design Debt and Code Debt could be used to indicate design problems by using existing code smell detectors. For the latter, the findings can be used as guidelines to avoid the introduction of TD and consciously reduce co-occurring TD, which would in turn reduce the cost of software maintenance.

The remainder of this chapter is structured as follows. Section 3.2 discusses the study objectives, the research questions, and provides details regarding the data collection and analysis. Section 3.3 reports on the results of our study and Section 3.4 discusses the results and their implications. Section 3.5 reports on the threats to the validity of our study and provides information about our dataset. After the discussion of related work in Section 3.6, Section 3.7 concludes the chapter.

3.2 Study Design

This empirical study was designed according to the guidelines of Runeson et al. (2012) and is reported according to the Linear Analytic Structure (Runeson et al.,

²According to the Tiobe Index, one of the best known indices of programming languages popularity, <https://www.tiobe.com/tiobe-index/>

2012).

3.2.1 Objectives and Research Questions

The goal of our study, described according to the Goal-Question-Metrics (GQM) approach (Van Solingen et al., 2002), is to “analyze Apache software systems written in Python for the purpose of investigating various types of technical debt with respect to the co-occurrence of these types, from the point of view of software developers in the context of open source software”. This objective is further refined into two research questions:

RQ1: Which types of technical debt frequently co-occur?

The goal of this research question is to analyze the types of technical debt that co-occur in the same file. The results can shed further light on understanding the ‘domino effect’ of particular types of associated debt, i.e., specific types of debt that usually lead to other types. This may draw developers’ attention to specific types of TD that tend to result in even more TD, and thus extra complexity and maintenance effort.

RQ2: How long is the interval between the appearance and disappearance of co-occurring technical debt issues?

This research question aims at analyzing the average time between the appearance and disappearance of the co-occurrences. Such information would provide additional insights to developers, further supporting the prioritization of remediation. For example, a short time-span (on average) between different co-occurring issues may strongly indicate that one causes the introduction or removal of the other.

3.2.2 Case Selection

To perform our analysis, we considered Python projects from the ASF as subject systems. These have high quality and long-term stability since they are managed by self-selected teams of technical experts³. The ASF includes 48 Python projects of different domains, sizes (up to 1,000 KLOC), activity (up to 10K commits to the master branch) and number of files (up to 700) on GitHub. To select systems among them, we used two main inclusion criteria:

1. The project must show up on the Apache Projects List⁴, which excludes Apache Incubator projects. Incubated projects are on a transition period to conform to Apache standards and, are therefore non-representative.

---

³<http://www.apache.org/foundation/how-it-works.html>

⁴<https://www.apache.org/index.html#projects-list>, visited in Sep. 2019

2. The project's main programming language must be Python, i.e., the largest number of files and source lines of code (SLOC) are written in Python.

Based on the criteria, we selected 20 Python projects⁵, which include 3682 Python source files from the ASF. The projects have an average of 1410 commits (median: 187; max: 10942), 124 files (median: 32; max: 865) and 17K SLOC (median: 4K; max: 126K). The majority of projects have a long history of commits (at least 3 years), which allows following the co-occurrence of technical debt issues over an extensive period. The majority also has a large number of files, which mitigates potential threats to the external validity (i.e. how far the sample represents the population).

To analyze the evolution of project files, the historical information of the project is required. To enable the study of long-lived software systems and standardize the data collection, we decided to take equally-spaced snapshots. In particular, a period of one week was used to guarantee the inclusion of sufficient revisions for each analyzed system. The same method was used in another research on TD evolution (Digkas et al., 2018).

3.2.3 Variables

Each unit of analysis comprises the tuple:

<file identification; snapshot time-stamp; TD information>

The *file identification* comprises the *project name* and *file path*. The *snapshot time-stamp* regards the date in which the weekly snapshot was taken. Finally, *TD information* regards the amount of open issues in a particular file and snapshot. Section 3.2.3 elaborates on the types of TD collected, while Section 3.2.3 explains the data collection procedure.

Technical Debt Identification

To perform our study, as aforementioned, we use SonarQube as the tool to detect technical debt. SonarQube defines a set of rules to detect various types of technical debt and classifies them into four severity levels based on impact and likelihood: *blocker*, *critical*, *major* and *minor*. The *minor* issues are trivial and commonly perceived as having little to no relevance (e.g., *Lines should not end with trailing whitespaces*) and could, therefore, bias the results⁶. Thus, we limited the severity level to *blocker*, *critical* and *major*. During analysis, SonarQube creates a new issue when a piece of code breaks one of the predefined rules.

⁵A detailed description of the projects can be found at <https://github.com/jieshanshan/SEAA2020/blob/master/Projects.md>

⁶<https://docs.sonarqube.org/latest/user-guide/rules/>, visited in Dec. 2019

Moreover, issues related to some rules (e.g., *Syntax error*) are essentially interpreter errors instead of technical debt; thus, we also excluded such rules. We clarify that we maintain rules that can be associated with changes between Python2 and Python3 (e.g., *The "print" statement should not be used*). After removing issues of minor severity and related to interpreter errors, we are left with a total of 42 rules⁷. These rules are mapped to categories (e.g. complexity, duplicated code) and then to the five TD types⁸, i.e., Design Debt, Code Debt, Test Debt, Documentation Debt and Defect Debt (Tan et al., 2020). These types were independently derived by two different studies, i.e. by Alves et al. (2014) and Li et al. (2015). The *TD information* variable for each unit of analysis is the count of issues for each of the 42 rules.

Data Collection

To perform our study, we started by cloning the GitHub repositories of the Python projects. Then, we wrote a script to: (1) extract the entire change history of each project; (2) group it by week, therefore defining the snapshots, and (3) submit the snapshots to SonarQube in chronological order. The weekly commits were then analyzed by SonarQube to identify the technical debt issues.

Finally, we use a second script to obtain all the information for each issue, i.e., which rule it breaks, when it appeared and disappeared in the file and the name of the file. With this information, we calculate the variables the comprise the TD information of each unit of analysis, i.e., the TD of each file for each snapshot.

Overall, we detected more than 23K issues related to 42 different rules of technical debt from over 45K commits of selected 20 Python projects.

3.2.4 Data Analysis

In this section, we first explain a method that is key to our analysis, namely *Association Rule Mining*. Subsequently we describe the steps necessary to analyze the collected data and answer the proposed research questions.

Association Rule Mining

This method is used for detecting an association between different objects in a set, by finding frequent patterns in a transaction database, relational database or any other information repository. In this chapter, we perform association rule mining based

⁷The detailed description of the rules can be found at https://github.com/jieshanshan/SEAA2020/blob/master/TD_Rules.csv

⁸The mapping of rules to categories and types is explained at https://github.com/jieshanshan/SEAA2020/blob/master/RuleType_map.pdf

on a well-known machine learning algorithm, namely Apriori (Agrawal et al., 1993). The same method is also used by Palomba et al. (2018a). Following the original definition by Agrawal et al. (1993), we let $TD = \{td_1, td_2, \dots, td_n\}$ be a set of n binary attributes composed by all types of technical debt and each file contains a subset of the types of TD .

In our study, the direction of the relation is defined as an implication of the form:

$$td_i \rightarrow td_j, \text{ where } td_i, td_j \in TD, \text{ and } i \neq j, \quad (3.1)$$

to represent the relationship that td_i appears before td_j in the same source file.

To answer each RQ, we introduce the following formula to measure association between different types of technical debt:

$$\text{confidence}_{td_i \rightarrow td_j} = \frac{\text{support}(td_i \wedge td_j)}{\text{support}(td_i)} \quad (3.2)$$

where $\text{support}(td_i \wedge td_j)$ is the probability of both td_i and td_j appearing in the same file and $\text{support}(td_i)$ is the percentage of files that contain td_i . This formula represents the likelihood of td_j appearing when td_i already exists in the same file, expressed as $td_i \rightarrow td_j$.

The drawback of this method in calculating the percentage of co-occurrences of td_i and td_j is that it might misrepresent the relationship, as it only accounts for how popular the issue td_i is but does not consider td_j . If the rule td_j is also appearing frequently in general, there will be a higher chance that a file containing td_i will also contain td_j , thus impacting on the value and accuracy of the relationship between co-occurring technical debt.

To account for the prevalence of co-occurrences, we use the following formula:

$$\text{lift}_{td_i \rightarrow td_j} = \frac{\text{support}(td_i \wedge td_j)}{\text{support}(td_i) \times \text{support}(td_j)}, \quad (3.3)$$

which can be interpreted as the probability of both td_i and td_j co-occurring in the same file while controlling for the popularity of both td_i and td_j . The interpretation of the results of this formula is as follows:

- if $\text{lift}_{td_i \rightarrow td_j} = 1$, it would imply that the probability of occurrence of td_i and td_j are independent of each other, thus no rule can be drawn involving them.
- if $\text{lift}_{td_i \rightarrow td_j} > 1$, the value gives the degree of interdependence between td_i and td_j , and makes that relationship potentially useful for predicting the consequent technical debt in the future. Greater lift values indicate stronger

associations.

- if $\text{lift}_{td_i \rightarrow td_j} < 1$, it means that presence of td_i has negative effect on presence of td_j , and vice versa.

Analysis Procedure

To answer **RQ1**, we use the information of the evolution per issue to find all the dates on which rule violations were introduced or removed, henceforth referred to as *debt-changed-date*. After that, we calculate the percentage for each TD rule on the 20 analyzed projects per *debt-changed-date* and use the average value $\text{support}(td_i)$ to represent how popular td_i is during the evolution of these projects.

Then, we use the percentage of files that contain both td_i and td_j to divide the percentage of files that only contain td_i to calculate $\text{confidence}_{td_i \rightarrow td_j}$ for each project on its *debt-changed-dates*. Similarly, we also get the average value as the final result for $\text{confidence}_{td_i \rightarrow td_j}$ to investigate how often the existence of one rule td_j relates to the occurrence of another rule td_i in the same file.

As $\text{confidence}_{td_i \rightarrow td_j}$ shows how popular a co-occurring relationship between td_i and td_j is, a low value of $\text{confidence}_{td_i \rightarrow td_j}$ means that a co-occurrence $td_i \rightarrow td_j$ is likely to be occasional and meaningless, or one of the debt rules (i.e., td_i or td_j) does not appear frequently during the evolution of Python projects. To avoid the analysis of meaningless relationships, we choose the top ten percent of the total $\text{confidence}_{td_i \rightarrow td_j}$ values.

Finally, we investigate how often the occurrence of one rule td_i in a source file is related to the introduction of another rule of technical debt td_j . It is worth noting that the previous results show how popular the co-occurring rules are. However, we still cannot be sure that there is a strong relationship between them, especially when td_j appears much more frequently than the other rules. In that case, we do not have strong evidence that the introduction of td_j is related to the introduction of td_i . To handle this research question, we select the valid co-occurrences with $\text{lift}_{td_i \rightarrow td_j} > 1$.

For **RQ2**, we analyze the pairs of co-occurring TD detected in **RQ1**. For each pair, we use all reported issues to calculate and analyze the interval of introduction and removal between two co-occurring issues found in **RQ1**. These variables are measured as the number of days between the moments when they are introduced (Formula 3.4) and removed (Formula 3.5) in the file:

$$\text{intro-interval}_{td_i \rightarrow td_j} = \text{intro}_{td_i} - \text{intro}_{td_j} \quad (3.4)$$

$$\text{remov-interval}_{td_i \rightarrow td_j} = \text{remov}_{td_i} - \text{remov}_{td_j} \quad (3.5)$$

Figure 3.1: Network of top 10% co-occurring rule violations

note that $\text{intro-interval}_{td_i \rightarrow td_j}$ differs from $\text{intro-interval}_{td_j \rightarrow td_i}$, and if td_i is introduced before td_j , the value of $\text{intro-interval}_{td_i \rightarrow td_j}$ is negative. For the removal of co-occurring issues, if only one issue (e.g., td_j) is removed and the other co-occurring issue (td_i) still exists in the file, we regard the removal time of the latter as an infinite value, i.e., the interval of removal is infinite ($\text{remov-interval}_{td_i \rightarrow td_j} \rightarrow \infty$). As a reminder, we are interested in both issues being removed instead of just one; this is because we are interested in associating the removal of one to the other, especially if removing one causes the removal of the other.

3.3 Results

3.3.1 RQ1: Which types of technical debt frequently co-occur?

We calculated the number of TD types (i.e., Design Debt, Documentation Debt, Test Debt, Code Debt, Defect Debt) that affected each file at the same period. We found that 90.3% (i.e., 3324 out of 3682) of the files were affected by at least two debt types. This signifies that Python files are predominantly afflicted by multiple TD issues.

To examine this phenomenon in more depth, we look at the numerous rules for the different TD types (see Section 3.2.3). Specifically, in the following we compare the confidence values of every pair of co-occurring rules. For that, we organized the

Figure 3.2: Distribution of the possibility of co-occurrence and lift values for the violations of 29 pairs of rules

confidence values of the 1722 pairs in bins of size 0.02. We found that the majority of the confidence values (72.6%) are under 0.02, i.e., **most of the associations tend to be occasional and meaningless**.

As discussed in Section 3.2.4, we choose the top ten percent of the total $\text{confidence}_{td_i \rightarrow td_j}$ values. Figure 3.1 shows a network describing those selected 172 top co-occurring associations (as aforementioned, the ID, types and descriptions of the rules can be found online⁹). Each node represents one TD rule, while the node label is marked with the rule ID number. The nodes are color-coded according to their TD type. Larger nodes denote higher popularity (i.e., chance of appearing in a file). The edges are directed according to the direction of the co-occurrence. From this figure, one can observe that some of the more central nodes regard Design Debt, while Documentation and Test Debt are also very central but with two and one nodes respectively.

To investigate how often the occurrence of td_i in a source file is related to the appearance of another rule td_j , we choose the co-occurrences with $\text{lift}_{td_i \rightarrow td_j} > 1$. Table 3.1 reports detailed information on the top 16 rules involved in these selected associations, denoting the type, ID number, definition and percentage of files that contain it. Figure 3.2 reports a bar chart comparing the lift values and the probability of the co-occurrences. We do not observe a correlation between them. This is mostly because the probability of the co-occurrence is greatly affected by the popularity of each rule, but lift value is not. In short, the likelihood of $\text{confidence}_{td_i \rightarrow td_j}$ does not reflect on the strength of the relationship and, thus, the examination of $\text{lift}_{td_i \rightarrow td_j}$ helps to normalize the co-occurrence by the popularity of both co-occurring rules.

To illustrate this through an example from Figure 3.2: $\text{confidence}_{T56 \rightarrow D57}$ is the most likely association since it displays the highest value (63.9%) among the 37

⁹<https://github.com/jieshanshan/SEAA2020>

Table 3.1: Mapping of highlighted rules to types of technical debt

Rule	Definition	S% ¹
T56	Lines should have sufficient coverage by tests	98.2
D14	Docstrings should be defined	87.8
D57	Source files should have a sufficient density of comment lines	62.8
C50	Lines should not be too long	39.0
S29	Cognitive Complexity of functions should not be too high	32.0
S58	Source files should not have any duplicated blocks	23.4
C42	Function names should comply with a naming convention	17.7
F21	The “print” statement should not be used	17.7
S24	Control flow statements should not be nested too deeply	17.2
C16	Sections of code should not be “commented out”	17.1
S20	Functions should not be too complex	15.3
S41	Collapsible “if” statements should be merged	14.6
C394	File should not use “import”	14.2
S40	Functions should not contain too many return statements	13.5
S47	Files should not have too many lines of code	10.1
C46	Functions, methods and lambdas should not have too many parameters	9.0

¹ support values, i.e., the percentage of files that contain the rule

co-occurrence associations. However, $\text{lift}_{T56 \rightarrow D57}$ is close to the median value, which may denote that both T56 and D57 appear in the majority of Python files. The percentage of files that contain one of these two rules is shown in Table 3.1. Therefore, it has indeed a high chance of simultaneously occurring in the same file.

The top four co-occurrences are between Design Debt issues regarding complexity, i.e., *cyclomatic complexity* (S20), *cognitive complexity* (S29) and *spaghetti code* (S24). As shown in Figure 3.2, $\text{confidence}_{S20 \rightarrow S29}$ is the highest one, i.e., 35.1% of files affected by S20 (*cyclomatic complexity*) are also affected by S29 (*cognitive complexity*), followed by $\text{confidence}_{S29 \rightarrow S20}$ (26.3%). This indicates that **cyclomatic complexity and cognitive complexity are strongly intertwined** during the evolution of Python files. In other words increase in cognitive complexity triggers increase in cyclomatic complexity and vice versa. Furthermore, the results suggest that **cyclomatic complexity (S20) and spaghetti code (S24) have a similar impact on each other**, i.e., similar probability of co-occurrence.

Regarding Test Debt (T56) we observed that it **tends to co-occur with other rule violations and appears in almost all files throughout their evolution**. However, all of the lift values that are related to Test Debt are close to 1, i.e., **the probability of insufficient test coverage for introducing other rule violations is reduced**.

Regarding Defect Debt, one rule (F21) out of the 15 available is involved in the frequently co-occurring associations. Moreover, the lift values between Defect Debt

Figure 3.3: Distribution of the possibility of co-occurrence and lift values for the violations of 27 pairs of rules

and Test Debt are higher than the other rules related to Design Debt and Code Debt in Figure 3.2. These observations show an indication that **incompleteness of software testing could be associated with an increase in Defect Debt**. In addition, issues related to F21 are caused by the update of the Python interpreter (from Python2 to Python3). Thus, the increase of test coverage may greatly support maintenance activities when the interpreter is updated.

Concerning the relationship between Test Debt and Code Debt or Design Debt, all of the lift values between them are near 1.01. Thus, there seems to be **no obvious relationship between the introduction of Test Debt and the increase in Code Debt and Design Debt**. In addition to rule T56, rule S29 (*Cognitive Complexity of functions should not be too high*), has the second largest number of co-occurrences. However, compared with T56, the lift values are higher than the others. Figure 3.2 shows that 70% of the top ten lift values are related to S29. Moreover, the majority of the other rules involved in these associations are also related to Design Debt, only two of them are related to Code Debt. Such evidence suggests strongly that **Design Debt issues are often associated with more problems that can be of similar nature (i.e. other Design Debt issues) or that can be caused by the same design decisions**.

3.3.2 RQ2: How long is the interval between the appearance and disappearance of co-occurring technical debt issues?

To answer this research question, we calculated the interval values ($\text{intro-interval}_{td_i \rightarrow td_j}$ and $\text{remov-interval}_{td_i \rightarrow td_j}$) for each pair of the frequently co-occurring associations found in RQ1 (Section 3.3.1). Note that if td_i is introduced before td_j , the value of $\text{intro-interval}_{td_i \rightarrow td_j}$ is negative. Similarly, if td_i is removed before td_j , the value of $\text{remov-interval}_{td_i \rightarrow td_j}$ is negative.

Figure 3.3(a) and Figure 3.3(b) show diverging stacked bar charts to visualize

the percentages that describe the distribution of intervals for the different pairs of co-occurring technical debt issues. The percentages show how often an issue td_i was introduced (Figure 3.3(a)) or removed (Figure 3.3(b)) before an issue td_j within a certain time-frame (i.e., one week, one month, one year and after one year). The time-frames are represented by the different shades of colors, ranging from “within one week”, to “more than one year”.

The main finding based on Figures 3.3(a) and Figure 3.3(b) is that **if co-occurring issues are not introduced within one week, then there is a negligible chance of them appearing within one month**; in most cases they will appear after at least one year. However, once introduced, we notice that **co-occurring issues tend not to be removed within weeks or months, but will likely survive at least one year** in the file. Combining these two findings, TD co-occurrence seems to happen either very quickly or not at all; but once it does happen, it is hard to get rid of. Furthermore, **co-occurring Design Debt issues seem to be repaid in a shorter time** as shown in Figure 3.3(b).

3.4 Discussion

3.4.1 Interpretation of Results

The phenomenon of co-occurring TD is widely spread in Python projects; this confirms a similar observation regarding Java projects (Palomba et al., 2018a). Moreover, all the Design Debt rules shown in Table 3.1 are not Python-specific and have also been detected in projects written in other programming languages, such as Java and JavaScript¹⁰. This indicates that these particular results do not apply to Python only. For example, we observe that rule S58 (*duplicated blocks*) co-occurs with S47 (*long file*), which is similar to what Yamashita et al. (2015) observed for Java, i.e., duplications appear together with smells related to size. Thus, **some co-occurring TD issues may apply also to Java**.

However, the rest of our results are limited to Python and can aid practitioners and researchers specifically for this language. To begin with, there tends to be a strong association between Design Debt issues and Code Debt issues. This result is supported by the high lift values observed in the investigation of RQ1 and the shorter removal intervals observed in the investigation of RQ2. It indicates that **investigating issues at code level can support assessment and reasoning about the design of a software-intensive system**.

Comparing the network in Figure 3.1 and the corresponding support values in Table 3.1, we notice that Design Debt issues are strongly interconnected. However,

¹⁰<https://rules.sonarsource.com/>

this is not uniform across all projects. For example, issues related to cyclomatic complexity (S20) and spaghetti code (S24) co-occur most frequently in a software development collaboration tool (BLOODHOUND), but those co-occurrences are less likely to appear in projects for messaging APIs or test suites (e.g., QPID). Thus, **projects in different domains show differences regarding which TD issues co-occur.**

Moreover, Design Debt issues not only seem to be strongly associated with one another (RQ1), but they also tend to be removed together and quickly (RQ2). For example, issues related to long files (S47) tend to be resolved together with other issues the fastest; this is probably because files are shortened when undertaking refactoring activities to address other (co-occurring) issues. These findings corroborate earlier findings Palomba et al. (2018a) that **Design Debt issues are generally removed together as a consequence of maintenance and evolution activities.**

We also found that, although Test Debt tends to co-occur with issues regarding other TD types, it has a lesser impact on the increase of Code Debt or Design Debt. These results are not in full accordance with related work, which suggest that Test Debt has a strong negative influence on maintenance (Spadini et al., 2018). A possible reason is that only one rule related to Test Debt was involved in our study. To further investigate this relationship, we calculated the Spearman correlation between the percentage of files that contain both types of debt (Test and Design Debt) and the percentage of files containing Design Debt only (see the corresponding values in Table 3.1). We found the correlation to be positively strong¹¹ ($\rho = 0.98$), which further confirms that **Test Debt has a lesser impact on the increase of Design Debt.**

Comparing the findings from the appearance and disappearance of co-occurring TD (Figures 3.3(a) and 3.3(b)), the percentages of co-occurring rule violations that are introduced within one week are much higher than those that are removed within one week, especially for Test Debt. Moreover, Test Debt is always removed later than others. These observations are possibly related to adopted practices, or lack thereof, as we were able to identify evidence of test-driven development in only one project (ALLURA). This indicates that, since it always appears earlier than the others, **Test Debt seems easier to incur, but also more difficult to resolve, possibly because it is perceived as less important.**

3.4.2 Implications for Researchers and Practitioners

Based on the interpretation of our results, we highlight that Design Debt issues may be associated with more similar problems; thus, practitioners should target them

¹¹We interpret the correlation coefficient according to Cohen (1988), i.e., strong correlation when $0.5 \leq |\rho| \leq 1$.

in earlier refactoring activities. On the contrary, fixing Test Debt can be postponed since it has a lesser impact on increasing the probability of introducing other TD issues and tends to be harder to resolve. However, one should be careful with such trade-offs as refactorings may lead to bugs and, without appropriate tests, also to catastrophic results.

The findings might help practitioners to estimate the risk level of potential TD issues and make decisions over design problems by using code-level insights. For example, the accumulation of issues such as cognitive complexity and code duplication in components could raise the risk of specific design problems emerging in the near future.

Researchers can use our results to guide the development of theoretical frameworks and associated research tools, as well as a basis for future studies. In particular, the results of frequently co-occurring TD can be used as an input to calculate TD indices that integrate metrics from individual issues with their co-occurrence information. In addition, TD tools can improve the prioritization of TD remediation by calibrating the weights for the different TD issues. For example, Design Debt issues can be assigned higher weights since they are more interconnected and repaid in a shorter time.

Our findings can also be used to warn developers about potential future co-occurring issues. For example, developers could be cautioned that incompleteness of software testing could be an indicator of increasing Defect Debt, especially Defect Debt issues caused by the update of the interpreter.

3.5 Threats to Validity

In the following, we discuss the threats to construct and external validity of the reported study, as well as reliability threats. We note that we do not analyze internal validity, since we do not seek to establish causal relations but to study the associations between various TD issues.

Construct validity pertains to the connection between the research questions and the objects of study. In this regard, the result of this study relies on SonarQube to detect TD issues. Although the tool has been widely used in both industry and academia, our interpretation of TD is limited to the tool's capabilities. Different tools could use different strategies to detect TD, which might lead to variations in the TD issues and, consequently, potential co-occurrence associations.

Another threat is related to how we analyze the evolution of projects. In particular, we analyze the change history on a weekly basis and, thus, the introduction and removal time might have a maximum error of one week. Moreover, projects may

receive new commits at different rates, which may affect the time-frame between the introduction or removal of co-occurring issues in different projects. To mitigate potential bias we use the Apriori method to identify the associations.

The analyses reported in this chapter were performed at file level, which may be affected by the high diffuseness of rules from a certain TD type (e.g. Test Debt), increasing the probability of co-occurring issues. Other studies have also performed analyses at file level to investigate co-occurrence (Yamashita and Moonen, 2013; Yamashita et al., 2015) but, unlike them, we introduced the lift value to further control for the popularity of co-occurring issues.

External validity concerns threats to the generalizability of our findings. The main threat here comes from the fact that we analyzed the evolution of 3862 files from 20 projects that are limited to the ASF and Python systems. Despite the credibility of the Apache Foundation and the diversity of its Python projects, our results may not fully represent the entire population of non-trivial Python projects. Furthermore, the set of rules considered in this study is not exhaustive and does not portray the complete set of TD related issues that may affect Python source code.

We emphasize that the size of the dataset is also limited by the use of SonarQube and its multi-version analysis: collecting the data required for this analysis can take a long time. In our case, analyzing 20 systems took more than two months of work on an Intel Core i7-5500U PC with 8GB of RAM.

Reliability threats were addressed by involving at least two researchers in both data collection and analysis. Also, samples of analyses output at the different steps were manually inspected for irregularities and alignment with the proposed study design.

3.6 Related Work

In recent years, the research community has extensively analyzed the effect of individual instances of TD (Sjøberg et al., 2013), while several studies also focused on the correlation between different items of technical debt. Some results indicate that code smells that co-occur in the same source file can intensify negative effects on software maintenance (Yamashita and Moonen, 2013; Yamashita et al., 2015). Palomba et al. (2018a) investigated six pairs of frequently co-occurring instances among 13 code smells, and found that code smells are generally removed together as a consequence of maintenance. Fontana et al. (2015) found that code smells co-occur and interact in many ways, which is confirmed in our study.

Several studies also focused on the association between different debt types. Abbes et al. (2011) found that the combination of two design smells, namely God

Class and Spaghetti Code, significantly reduces program comprehension when compared to only one. The results suggest that developers could deal with one design smell but the combination of different design smells should be avoided through detection and refactoring. Spadini et al. (2018) investigated the relationship of six test smells and their co-occurrence with software quality. They found that detecting test smells is important for the effectiveness of defect detection. Bavota et al. (2015) studied the relationship between test smells and their effects on software maintenance, observing that these smells are highly diffused (Bavota et al., 2015). However, we did not confirm that Test Debt has a strong negative impact on issues related to software maintenance, i.e., Code Debt or Design Debt.

Unlike the aforementioned studies, which focused on one or two debt types only, we analyzed the co-occurrence between five types of debt.

3.7 Conclusion

This chapter reports on an empirical study that investigated the relationship between co-occurring technical debt types. We performed our study by analyzing the multi-annual evolution of TD in 20 Python projects (approx. 28,200 commits) from the ASF and identifying TD based on 42 rules defined in SonarQube. Moreover, we mined co-occurring TD based on a well-known machine learning algorithm, namely Apriori, in 3862 files.

We found that the phenomenon of co-occurring TD is highly spread, i.e., more than 90% of Python files have been affected by at least two types of TD at the same time. In particular, from the 1722 possible associations between rules, 48 pairs frequently co-occurred in files, from which 27 are more strongly associated with each other. Among them, Design Debt issues more commonly co-occur with issues of similar nature, while Documentation Debt and Test Debt are the most prevalent in the studied projects.

Furthermore, we found that co-occurring issues tend to be introduced within the same week. However, after one of the co-occurring issues is removed, the second one survives for much longer (often around a year). Moreover, co-occurring Design Debt issues seem to be repaid in a shorter time, while most of the Test Debt issues are introduced before the other debt types but it is also the last to be repaid. These findings may indicate that developers are more aware of Design Debt compared to the other types, and that Test Debt may be regarded as a difficult type of debt or with less priority.

This chapter focused on the co-occurring technical debt regarding its types and survival time, and such observation came from the investigation of TD remedia-

tion (i.e., Chapter 2). Another observation that stemmed from Chapter 2 was that a considerable portion of the TD items were fixed by the same developers who introduced them. This leads to the next chapter, which reports on the self-fixed technical debt phenomenon.

Based on:

Jie Tan, Daniel Feitosa, Paris Avgeriou, (2022) *“Does it matter who pays back Technical Debt? An empirical study of self-fixed TD”* in Information and Software Technology, vol.143,
DOI:10.1016/j.infsof.2021.106738

Jie Tan, Daniel Feitosa, Paris Avgeriou, *“An Empirical Study on Self-Fixed Technical Debt”* in Proceedings of the 3rd IEEE/ACM International Conference on Technical Debt (TechDebt '20), Seoul, Republic of Korea, pp.1–12, DOI:10.1145/3387906.3388621

Chapter 4

Does it matter who pays back Technical Debt? An empirical study of self-fixed TD

Abstract

Technical Debt (TD) can be paid back either by those that incurred it or by others. We call the former self-fixed TD, and it can be particularly effective, as developers are experts in their own code and are well-suited to fix the corresponding TD issues. The goal of our study is to investigate self-fixed technical debt, especially the extent in which TD is self-fixed, which types of TD are more likely to be self-fixed, whether the remediation time of self-fixed TD is shorter than non-self-fixed TD and how development behaviors are related to self-fixed TD. We report on an empirical study that analyzes the self-fixed issues of five types of TD (i.e., Code, Defect, Design, Documentation and Test), captured via static analysis, in more than 44,000 commits obtained from 20 Python and 16 Java projects of the Apache Software Foundation. The results show that about half of the fixed issues are self-fixed and that the likelihood of contained TD issues being self-fixed is negatively correlated with project size, the number of developers and total issues. Moreover, there is no significant difference of the survival time between self-fixed and non-self-fixed issues. Furthermore, developers are more keen to pay back their own TD when it is related to lower code level issues, e.g., Defect Debt and Code Debt. Finally, developers who are more dedicated to or knowledgeable about the project contribute to a higher chance of self-fixing TD. These results can benefit both researchers and practitioners by aiding the prioritization of TD remediation activities and refining strategies within development teams, and by informing the development of TD management tools.

4.1 Introduction

When Technical Debt (TD) results increasingly in extra effort during maintenance and evolution, it needs to be (at least partially) paid back, e.g., through refactoring. Previous studies have shown that a large percentage of source code Technical Debt (TD) is indeed paid back (Digkas et al., 2018; Maldonado et al., 2017). There is however, an elementary distinction between two cases of TD repayment, depending on whether the debt is paid back by the developer who introduced it or by others. We call the former *self-fixed TD*, and it is significant, because developers are familiar with their own code, and may perform the repayment differently than others. On the contrary, we call the latter *non-self-fixed TD*. Of course, remediation of TD incurred by others also takes place systematically. For example, developers that join an open source project are ideally pointed to easier tasks, which can involve update of documentation or simple code hacks (Steinmacher et al., 2019). This helps the newcomers to build know-how and get used to the team’s operations.

In spite of the intriguing nature of the mentioned distinction, the phenomenon of self-fixed technical debt has not been thoroughly studied so far. Investigating this can help identify good and bad practices in TD remediation, especially when comparing TD that is self-fixed and TD that is not. In particular, there are two aspects to study with regards to self-fixed TD: the phenomenon itself, and how human factors are related to it.

Studying the phenomenon itself may help to understand which types of TD are self-fixed and which ones are not. This can provide an insight on how developers prioritize the remediation of different types of TD, either their own or of others. Moreover, it may highlight cases where TD lives long in systems before it gets fixed and thus forces developers to pay high amounts of interest. Such findings can also indicate how developers generate value (measured as improved quality) by managing their own TD. For example, learning what types of TD developers are more keen to self-fix or to self-fix faster may hint towards practices to boost software value creation.

Studying human factors is also important as they can have a significant impact on software quality. For example, a strong feeling of ownership over a piece of code might increase the pride of accomplishment and contribute to delivering code with higher quality and fewer defects (Rahman and Devanbu, 2011). Investigating such factors can further explain why developers self-fix TD differently, and why some projects are more likely to have TD that is not fixed. To the best of our knowledge, only a few studies have addressed the relationship between human factors and technical debt management (Amanatidis et al., 2017; Alfayez et al., 2018); among them, there is no research that focuses on how human factors influence developers to self-

fix TD.

To the best of our knowledge, this chapter reports on the first study to exclusively focus on self-fixed technical debt. First, we investigate the extent of this phenomenon, as well as the types of TD that are more likely to be self-fixed and the survival time of self-fixed TD, especially compared to non-self-fixed TD. Second, we examine the relationship between human factors related to the effort of developers and development teams against the self-fixing rate, together with exploring the potential aspects that may influence developers in fixing their own TD.

To perform the study, we analyzed more than 17K commits from 20 Python projects and 27K commits from 16 Java projects of the Apache Software Foundation and investigated five types of TD, namely Code Debt, Defect Debt, Design Debt, Documentation Debt and Test Debt; these five types are defined by Alves et al. (2014) and Li et al. (2015). We used SonarQube 7.0 to detect and measure TD by identifying violations of a number of rules, which correspond to the different types of TD. The justification for the choice of projects, language and tool is elaborated in Sections 4.3.2 and 4.3.3.

The results indicate that the phenomenon of self-fixed TD is prevalent, as about half of the fixed issues are self-fixed. In addition, Defect Debt and Code Debt receive more attention from developers who introduced them, which may indicate that these developers are more keen to pay back their own TD when it is related to lower code level issues (e.g., compared to Design Debt and Documentation Debt). Moreover, there is no significant difference in the survival time between self-fixed and non-self-fixed issues.

Looking at project characteristics, the likelihood of TD issues to be self-fixed is negatively correlated with project size, the number of developers and total number of issues. Despite that, having developers who are more dedicated to or knowledgeable about the project (e.g., by contributing more commits) may positively impact the probability of the issues to be self-fixed. However, the longer the developers are involved in a project, the less often they self-fix TD.

Summarizing our findings, we have explored the phenomenon of self-fixed technical debt and how human factors are related to how developers self-fix technical debt. Broadly, the contribution encompasses the following items:

- This is the first study to focus on how developers pay back their own technical debt in a comprehensive way, and looks at two of the most popular programming languages, i.e., Python and Java.
- The study provides a comparison of the self-fixed and non-self-fixed TD items in terms of number, type of debt, and survival time.
- The study provides insight into the circumstances (e.g., project size and the

number of developers) that would improve the likelihood of developers self-fixing technical debt, which can help predict project quality.

The remainder of this chapter is structured as follows. Section 4.2 presents the discussion of related work. Section 4.3 reports on the study objectives, the research questions, and provides details regarding the data collection and analysis. Section 4.4 illustrates the results of our study and Section 4.5 discusses the results in depth and their implications. Section 4.6 reports on the threats to the validity of our study and Section 4.7 concludes the chapter.

4.2 Related work

The body of knowledge in TD management has grown substantially over the years. We are particularly interested in work pertaining to TD remediation and human factors related to it.

Significant research on TD remediation has focused on fixing code smells, which is a very popular sub-type of design debt (Li et al., 2015). In this context, Chatzigeorgiou and Manakos (2010) found that the amount of code smells increases over time and Tufano et al. (2017) found that some smells could survive for a long time. Focusing on maintenance activities, Palomba et al. (2018b) found that the removal of code smells is beneficial for code change-proneness most of the times. Sjøberg et al. (2013) identified that the 12 analyzed code smells have no significant effects on maintenance effort. Finally, Digkas et al. (2018) observed that a large percentage of TD, including code smells, is paid back during software evolution.

Another aspect of TD remediation regards the concept of self-admitted technical debt (SATD), i.e. developers admitting incurring TD (e.g. in source code comments). In this context, Potdar and Shihab (2014) extracted SATD based on source code comments and found that developers introduce and remove SATD throughout all development activities, i.e., they do not only introduce or remove SATD during the beginning or end of their development activities. Other studies found that the majority of SATD is removed and often by the same developer who introduced it (Bavota and Russo, 2016; Maldonado et al., 2017). However, Potdar and Shihab (2014) also noticed that developers with more experience tend to introduce more SATD. In Section 4.5.1, after presenting our results, we discuss how these observations compare to our findings. Furthermore, Wen et al. (2020) manually analyzed 500 quick remedy commits, i.e., the commits that fix issues introduced by the same developers as the result of omitted code changes or errors in the previous commit. Such commits were identified by looking at their objective. During the analysis, Wen et al. found 49 remedy commits related to software documentation, among which 16 are related

to code comment (including missed removal of SATD instances).

Unlike the studies mentioned in the previous paragraphs, which considered one or two types of source code technical debt, or SATD, we analyzed five common types: Code, Defect, Design, Documentation and Test Debt in source code. Among these five types of TD, one previous study has shown that developers mostly considered Design Debt, Test Debt and Defect Debt (Codabux and Williams, 2013). Regarding TD remediation, we focused on who fixed TD, i.e., whether it was fixed by the developer who introduced it or not. In particular, we investigated the types and survival time of TD in a comparison between self-fixed and non-self-fixed TD.

Since software development is the result of teamwork, factors related to the development team can have a significant impact on software quality. For example, casual developers are likely to introduce quality-related problems of greater severity and in greater quantity (Lu et al., 2016). Thus, investigating the factors related to developers and development team may further explain why the self-fixing rates vary among different projects. In our study, such factors include the number of developers and the frequency of their modifications.

Regarding the relationship between TD and human factors, Amanatidis et al. (2017) investigated to what extent seniority relates to a tendency to accumulate more TD. Although they found a negative association, the evidence is still inconclusive. In addition, they discovered that 80% of the most TD-incurring developers have low project-related experience. In a separate study, Alfayez et al. (2018) investigated how different developers and developer characteristics, such as developer's seniority, are related to the introduction of TD in 38 Apache Java systems. They found that developer's seniority are negatively correlated with the amount of introduced TD. In contrast to these studies, we focused on remediation of TD rather than its introduction, and in the context of self-fixed TD.

There are also external factors that can affect how developers remediate technical debt, such as using tools. For example, Gilson et al. (2020) found that junior developers tend to use static analysis tools to value composite quality indicators (e.g., maintainability, reliability in SonarQube), even if they do not fully understand their meaning. However, Ernst et al. (2015) conducted a survey with 1,831 participants, primarily software engineers and architects, and found that the lack of tool support for accurately managing and tracking architectural sources of debt is a key issue and remains a gap in practice. Thus, the findings of how developers remediate their own TD can be used to improve the current tools.

4.3 Study design

This empirical study was designed based on the guidelines of Runeson et al. (2012) and is reported according to the Linear Analytic Structure (Runeson et al., 2012).

4.3.1 Objective and Research Questions

The goal of our study is to investigate self-fixed technical debt with respect to the extent of this phenomenon, its different types and survival time, and its association with human factors by analyzing Python and Java projects.

Based on this goal, this study will examine the following five research questions:

RQ1: How much technical debt is self-fixed in open source projects?

Ideally, a developer who introduces technical debt to source code is also the best candidate to pay it back in the future. Since developers have the best understanding of their own code, they are more likely to know the causes of technical debt they incurred and have potential remediation strategies. This research question aims to investigate to what extent developers indeed take care of their own technical debt and also investigate how project characteristics, i.e., SLOC, the number of commits and developers, and the total number of issues, relate to the likelihood that the contained issues to be self-fixed. Such findings can be used to further explain how developers pay back TD in projects with different characteristics, e.g., learning whether they are more likely to fix a TD item by themselves or wait for other developers to address it.

RQ2: Which type of technical debt is more likely to be self-fixed?

Different issues belong to different types of technical debt and require different amounts of effort to be resolved. Therefore, this research question examines which issues and their corresponding TD types are more likely to be fixed by the developers who introduced them. The answer to this research question will shed further light on how developers consider their own technical debt by allowing to look at how the type of TD plays a role on what extent TD is fixed. For example, some issues may be associated with higher risks and thus be regarded as important to fix by the developers that introduced them. Or the opposite, some issues may be regarded as ‘low hanging fruits’, i.e. easy to self-fix. In addition, software components containing issues that are mostly fixed by others could be inspected to ensure proper documentation to support future maintenance activities.

RQ3: How long does self-fixed technical debt survive during the evolution of a system?

This research question focuses on investigating if self-fixed issues are addressed faster than the issues that are introduced by other developers, and comparing which types of TD issues are likely to be self-fixed in a shorter time. Analyzing the survival time of self-fixed technical debt during the evolution of each system helps us better understand how developers prioritize technical debt repayment, and how much interest is paid accordingly. Since developers tend to be more familiar with their own code, one would expect that developers can address their own TD issues in a shorter time.

RQ4: How does the number of developers and the frequency of their modifications relate with the likelihood of TD issues to be self-fixed?

Previous research has shown that not only human aspects related to individual developers, but also the collaboration of team members can have a significant impact on software quality (Lu et al., 2016). Similarly, we explore not only the impact of individual developer behavior on the likelihood that the issues to be self-fixed, but also team behavior. In particular, we look into how many developers maintain a given file and how many times a file has been modified by all the participating developers. In the former case, a higher number of developers maintaining a file/module at the same time (assuming that other factors are constant) may be associated with a higher chance of issues being fixed by developers that did not introduce them in the first place. In the latter case, the more a file changes, the more likely an issue is to be self-fixed.

RQ5: Who (self) fixes technical debt?

Developers have different ranges of involvement and contribution within a system. In general, developers with a high involvement are more familiar and experienced with their projects (Rahman and Devanbu, 2011). This research question aims at investigating the possible factors that drive some developers to have a higher chance to self-fix TD than others, by looking at *seniority* (i.e., how long they have been working on the project), *file count* (i.e., how many files they have involved in), *commit count* (i.e., how many commits they have authored) and *commit size* (i.e., how many lines of code (LOC) they have contributed on average per commit).

4.3.2 Case Selection

For the purpose of this study, we selected Python and Java projects from the Apache Software Foundation (hereafter referred to as the Apache ecosystem) as subject sys-

tems. Python and Java are currently ranked as two of the top three most popular programming languages¹. While our earlier study (Tan et al., 2020) solely focused on projects written in Python, we expanded this study to include Java projects, improving the external validity of our findings.

The Apache ecosystem is one of the largest open source foundation, with more than 8,000 Apache code committers². In addition, the ecosystem has 55 Python and 1,000 Java projects on GitHub, which contain different domains, sizes, activity and number of files. Finally, Apache projects have long-term stability³: a team of self-selected technical experts manages each project by following Apache-wide meritocratic rules, and an incubator filters projects based on their likelihood of becoming a successful community.

To select systems among the Apache projects on GitHub, we used four inclusion criteria:

1. The project must show up on the Apache Projects List⁴, which excludes Apache Incubator projects. Incubated projects are on a transition period to conform to Apache standards and are, therefore, non-representative.
2. The project's main programming language must be Python or Java, i.e., the largest number of files and source lines of code (SLOC) are written in Python or Java.
3. The project must involve at least two developers.
4. The project must have at least 10% of its issues fixed. A fixing rate lower than 10% may indicate poor quality and decreased attention on maintenance.

Analyzing the entire history of commits for some projects is prohibitive in terms of the required computational power and time. Thus, to still guarantee the inclusion of sufficient revisions for long-lived software systems, we decided to use at least the first 2,000 commits for each project. We particularly focus on the first 2,000 (rather than for example the last 2,000) since it facilitates determining the authors of commits that introduce TD. For projects that have less than 2,000 commits, we considered their entire history.

Based on the criteria, we selected all Python projects that fit these criteria at the time of data collection (i.e., 20 projects). Compared to Python the number of potential Java projects is vast; thus, we randomly selected 16 projects. The selected

¹According to the Tiobe Index, one of the best known indices of programming languages popularity, <https://www.tiobe.com/tiobe-index/>

²<https://www.apache.org/>, visited in November 2020

³<http://www.apache.org/foundation/how-it-works.html>, visited in November 2020

⁴<https://www.apache.org/index.html#projects-list>, visited in September 2020

projects partially represent the diversity of the Apache ecosystem in terms of SLOC, number of commits and developers. We note that although one Python project has two developers, 17 projects have at least five developers (median: 9; average: 18; max 85). Moreover, among the 16 Java projects 13 have at least 20 developers (min: 15; median: 55; average: 89 max: 319). Also, the majority of the projects have a long history of commits (at least 3 years), which allows to investigate the survival time of self-fixed technical debt over an extensive period (Marinescu, 2012).

4.3.3 Variables and Data Collection

In this section, we describe the sets of variables necessary to answer the research questions, as well as the necessary tooling and the major steps of the data collection process. In particular, each unit of analysis comprises the tuple:

<snapshot identification; snapshot information; TD issues>

The *snapshot identification* comprises the *project name* and *commit hashcode*. The *snapshot information* encompasses the *developer's name*, *developer's email*⁵, *time-stamp* (i.e., date of commit), and list of files (added, removed and modified). *TD issues* concern the amount of self-fixed and non-self-fixed issues in a particular snapshot. Data collection is comprised of three main steps.

Step 1: Technical Debt Detection

To perform our study, we use the tool SonarQube (Campbell and Papapetrou, 2013) to detect technical debt. There are two main reasons for choosing SonarQube: (1) it can track the evolution of technical debt by analyzing multiple versions of projects; (2) it is being widely used in both industry⁶ and for research purposes (Digkas et al., 2018; Marcilio et al., 2019; Lenarduzzi et al., 2019; Dale and Izurieta, 2014; Fontana et al., 2016). SonarQube defines a set of rules to detect various types of technical debt and classifies them into four severity levels: *blocker*, *critical*, *major* and *minor*. We do not consider *minor* issues, since many of them, e.g., *Lines should not end with trailing whitespaces*, are trivial and have low impact and likelihood⁷. Moreover, some developers might not treat issues of *minor* severity as technical debt (Digkas et al., 2018). During the analysis, SonarQube creates a new issue when a piece of code breaks one of the predefined rules and also assigns a time estimate of how long it would take to resolve it.

Based on the selected severity levels, SonarQube detects 49 and 52 rules in Python and Java projects, respectively. To simplify the organization of the tech-

⁵For privacy reasons, we do not disclose the developers' name and email in our replication package

⁶<https://www.sonarsource.com/customers/>, visited in September 2020

⁷<https://docs.sonarqube.org/latest/user-guide/rules/>, visited in September 2020

nical debt issues, we grouped the rules into five categories, i.e., Code Debt, Defect Debt, Design Debt, Documentation Debt and Test Debt, according to the classification given by Alves et al. (2014) and Li et al. (2015). In addition, we also filtered out rules that could not be meaningfully or directly mapped into one of the studied TD types. For example, we did not consider the FIXME rule since it can fit multiple TD types. Based on the selected severity levels and the selection, SonarQube detects 49 and 52 rules in Python and Java projects, respectively. The description of the rules and their classification can be found in Table B.4 (shown in Appendix A.1).

Based on the above, the variables that comprise the *TD issues* for each unit of analysis are: (a) the count of self-fixed and non-self-fixed issues for each type of debt, calculated as the sum of the issues for all rules mapped to that type, (b) the count of self-fixed and non-self-fixed issues for each rule, and (c) the remediation time for each self-fixed and non-self-fixed issue.

Step 2: Snapshot Data Extraction

For this step, we cloned the selected Apache Python and Java projects from GitHub.

Then, we used a script to: (1) extract the entire change history of each project; (2) reserve at least the first 2,000 consecutive commits per project, therefore defining the snapshots (see Section 4.3.2); and (3) submit the snapshots to SonarQube in chronological order. As mentioned before, for each snapshot, we recorded *project name*, *commit hashcode*, the changed files, and the *developer* who is the author of the commit, including their *name* and *email*. Furthermore, we only analyzed the commits of the main branch of the studied systems, since the changes are inspected responsibly by the project's core team (Kalliamvakou et al., 2014) and we focus on TD that has been repaid in the final product. We acknowledge that there are limitations with the use of such tools (e.g., thresholds associated with some detection rules), which are discussed in Section 4.6.

Step 3: Self-Fixed Technical Debt Identification

When analyzing multiple revisions of a system, SonarQube tracks the detailed information of issues that are fixed during the system evolution. For example, the first commit that contains the issue is considered as the commit that introduced it; similarly, the commit in which the issue disappeared is considered as the one that fixed that issue.

According to the documentation of SonarQube⁸, issues that are considered to be fixed could be involved in one of two situations: (1) issues have been corrected (i.e., the issues are truly fixed); or (2) the file is no longer available (removed from the project). The latter would be false positives in our study, as it is doubtful whether the

⁸<https://docs.sonarqube.org/latest/user-guide/issues/>, visited in September 2020

developers actually aimed at fixing the technical debt by deleting a file. To handle this problem, we filter out the issues that disappeared due to file deletion. The same method was used in some previous studies (Digkas et al., 2018; Tan et al., 2021).

To identify whether an issue is fixed by the same developer who introduces it, we search the *snapshots* to find the *developer* who introduced and fixed that issue. Furthermore, developers might submit a commit using a different email (e.g., work or personal) or version of the name (e.g., with acronyms). Thus, we compare developers' name and email address, considering it to be the same person when either the name or email address is the same.

4.3.4 Data Analysis

To answer **RQ1**, we compare the number of TD issues that were self-fixed and non-self-fixed (i.e., fixed by another developer) for each project p . In particular, we calculate the *self-fixing rate* per project, i.e., the percentage of TD issues that are fixed by the same developers in each project, which is measured as:

$$self\_fixing\_rate_p = \frac{count\_self\_fixed_p}{count\_fixed_p} \quad (4.1)$$

where $count\_self\_fixed_p$ is the number of self-fixed TD issues in project p and $count\_fixed_p$ is the total number of issues that have been fixed in the same project.

Furthermore, we use a generalized linear mixed model (GLMM) (McCulloch and Neuhaus, 2005) to investigate the relationship between the likelihood of an issue being self-fixed and several project characteristics: the number of commits, number of developers, SLOC, and the total number of issues.

For **RQ2**, we calculate the *self-fixing rate* per rule, i.e., the percentage of TD issues pertaining to rule r that have been self-fixed in all projects:

$$self\_fixing\_rate_r = \frac{\sum_{p=1}^n count\_self\_fixed_{p,r}}{\sum_{p=1}^n count\_fixed_{p,r}} \quad (4.2)$$

where n is the number of projects (i.e., 36), $count\_fixed_{p,r}$ is the amount of issues that have been fixed for rule r in project p and $count\_self\_fixed_{p,r}$ is the number of issues for rule r in project p that are self-fixed.

Subsequently, we compare the difference of the self-fixing rate for each debt type between Python and Java projects with Wilcoxon Rank Sum tests and calculate Cliff's Delta Effect Size. We also investigate which types of debt are more likely to be self-fixed and evaluate the significance of the observed differences by conducting a Scott-Knott Effect Size Difference test.

For **RQ3**, we calculate and analyze the survival time of each TD issue. This variable is measured as the number of days between the introduction of an issue and the moment when it is fixed in the source code. We use the Kaplan-Meier method to analyze and compare the survival time of the self-fixed and non-self-fixed issues.

To answer **RQ4**, we extract the detailed commit information to estimate how many developers maintain a given file f , i.e., $effort\_sharing_f$. This value is extracted by calculating the amount of developers that changed a file f from the time it was created to the last analyzed commit.

Using $effort\_sharing_f$ alone to represent effort sharing may incur some bias. For example, we hypothesize that the fewer developers contribute to the file⁹, the more likely they will modify their own code, implying a higher chance of fixing their own issues in that file. However, if these developers only make a few changes to the file, TD issues that were introduced by themselves may still remain in the project. Thus, investigating the number of changes that developers made to a file is also important. For that, we calculate an adjusted effort sharing using the following formula:

$$effort\_sharing\_adj_f = \frac{\sum_{d=1}^{count\_dev_f} change\_count_{d,f}}{count\_dev_f} \quad (4.3)$$

where $count\_dev_f$ is the amount of developers that changed a file f from the time it was created to the last analyzed commit; $change\_count_{d,f}$ is how many times a developer d changed one file. To assess the association between these two variables (i.e., $count\_dev_f$ and $change\_count_{d,f}$) and self-fixing, we build and examine a GLMM where the likelihood of an issue being self-fixed is predicted based on the two mentioned factors.

For **RQ5**, to investigate the association between independent variables related to developers and the likelihood of a developer to self-fix TD, we first introduce four variables.

- The *seniority* represents the number of days a developer has been involved in a project, and is based on Eyolfson et al. and Alfayez et al. Eyolfson et al. (2014); Alfayez et al. (2018). To calculate the seniority of each author, we first look for the author's last commit date in the analyzed commits and then subtract the author's first commit date.
- The *file count* refers to the number of files that a developer changed in throughout the analyzed history.

---

⁹By 'file contribution' we mean that a commit contains a change (i.e., line addition, removal or edit) to a given file.

- The *commit count* refers to the number of commits that a developer authored throughout the analyzed history.
- The *commit size* refers to the average number of LOC that a developer contributed per commit throughout the analyzed history.

Similar to the previous research questions, we build a GLMM to assess the relationship between the developer-related variables and the likelihood of issues being self-fixed by them.

4.4 Results

4.4.1 How much technical debt is self-fixed in open source projects?

This study required the mining of over 17K commits from 20 Python projects and 27K commits from 16 Java projects. After filtering out issues that were fixed due to file deletion (see Section 4.3.3), 54K Python issues and 71K Java issues have been selected in total, among which 35K and 34K, respectively, were fixed.

Table 4.1: Details of the 20 Python and 16 Java Projects

	Python			Java			All	
	Median	Average	Max	Median	Average	Max	Median	Average
Commits	337	853	2443	1454	1760	3799	1222	1227
SLOC	5307	14983	74658	87637	110881	316552	39043	56242
Developers	9	18	85	55	89	319	21	48
Total Issues	542	2679	24931	3518	4994	9904	2686	3723
Fixing Rate	48.85%	51.91%	86.34%	39.78%	44.44%	81.08%	48.86%	49.06%
Self-fixing Rate	67.21%	65.54%	100.00%	28.99%	33.26%	93.86%	47.64%	50.84%

Table 4.1 provides the following details about the 36 selected projects: the number of commits, SLOC, the number of developers involved in our study (i.e., in the analyzed commits), the absolute number of total issues, fixing rate, and the rate of the self-fixed issues. As shown in Table 4.1, Python projects seem to have a higher fixing rate and self-fixing rate than Java projects in general. Especially, about two thirds of the fixed Python issues are fixed by the same developers who introduced them, while this proportion is only one third in Java projects. Considering all projects together (Python and Java), almost half of the issues have been fixed, and among them, about half of the fixed issues were self-fixed. The results indicate that

about half of the technical debt seems to be paid back by the same developers who introduced them.

From Table 4.1, we also observe that the projects have different characteristics, i.e., the number of commits, developers, SLOC, the number of total issues, which might affect the self-fixing rates of different projects. To estimate how these characteristics relate to self-fixing, we build a generalized linear mixed model (GLMM) (McCulloch and Neuhaus, 2005), where the dependent variable captures, for each TD issue, whether it is self-fixed or non-self-fixed. The random effect is represented by the project ID, and helps to account for differences between projects in terms of the mentioned characteristics. In addition, to properly interpret the importance of each variable in the model, we use min-max normalization to normalize variable values, within each characteristic, in the interval $[0,1]$ ¹⁰. Moreover, we use the whole dataset from Python and Java projects to perform the analysis because we intend to build an explanatory model, not a predictive model.

To avoid multi-collinearity, we use the R function *redun*, of the *Hmisc* package (Harrell, 2019), for removing redundant variables. The function *redun* removes variables stepwise, starting from the most predictable one until no variable can be predicted with an adjusted R^2 greater than a given threshold, i.e., 0.8 in this study. The result of executing *redun* shows that there is no redundant variable. Then, we build the generalized linear mixed model using the function *glmer* of the *lme4* R package (Bates et al., 2007). Table 4.2 reports the results of the mixed-effect regression model. The top part of the table reports the model diagnostics, i.e., Akaike Information Criterion (AIC), Bayesian Information Criterion (BIC), log likelihood, deviance and degree of freedom residuals, the scaled residuals and the random effects. The bottom part of the table reports the odds ratio (OR), estimate, standard error, z-value and *p*-value¹¹.

As shown in the table, SLOC, the number of developers and the total number of TD issues have a statistically significant effect on the likelihood of an issue being self-fixed. The result indicates that **the size of the project, the number of developers involved in it, and the overall quality of the project (i.e., the total number of TD issues in this study) are associated with whether TD items are self-fixed, while the number of commits has no significant relation.**

In addition, Table 4.2 shows that the increase of SLOC, number of developers and number of total issues can decrease the likelihood that the issue will be self-fixed (i.e., $OR < 1$)¹². The results suggest that **as a project evolves, growing in size,**

¹⁰This is done by subtracting the minimum and dividing by the difference between the maximum and minimum, similarly as was performed in Iammarino et al. (2021)

¹¹Among these values, the *p*-value indicates whether the factor is statistically significant or not (for a significance level of 95%)

¹²OR is given by e^{c_i} where c_i is the coefficient of the *i*th factor for a logistic regression model. An OR

Table 4.2: Results of the generalized linear mixed model

Diagnostics					
	AIC	BIC	logLik	deviance	df resid.
	74950.6	75005.5	-37469.3	74938.6	68945
Scaled residuals					
	Min	1Q	Median	3Q	Max
	-11.338	-0.591	-0.510	0.912	4.368
Random effects					
Groups name	Variance	Std.Dev.			
Project ID (Intercept)	3.944	1.986			
Quality metrics					
Metric	OR	Estimate	Std.Error	z-value	p-value
(Intercept)	5.085	1.626	0.320	5.082	<0.001
SLOC	0.115	-2.164	0.600	-3.606	<0.001
Commits	1.318	0.276	0.483	0.573	0.567
Developers	0.138	-1.981	0.485	-4.082	<0.001
Total Issues	0.033	-3.426	0.501	-6.844	<0.001

number of developers and total issues, the likelihood of its TD issues to be self-fixed tends to decline.

4.4.2 Which type of technical debt is more likely to be self-fixed?

Figure 4.1 shows a series of box plots depicting the distributions of self-fixing rates of the five studied TD types among the 20 Python and 16 Java projects, together with the average and median values, and the number of fixed issues (bottom of figure). According to Figure 4.1, the self-fixing rate for each debt type in Python projects is nearly double compared to Java projects. To further evaluate the significance of the difference for each type of debt, we calculated the Wilcoxon Rank Sum test (Armstrong, 2014) and Cliff's Delta Effect Size (Cliff, 2014) on the self-fixing rates of the Java and Python projects, since we cannot assume the rates are normally distributed. Table 4.3 shows the results of the statistical tests, which reveal that except for Test Debt, the difference between the self-fixing rates of each type in Python and Java projects is significant (i.e., p -value < 0.05) and with a medium effect size¹³ at least.

¹³>1 indicates that an increase of variable increases of OR times the chances of an issue to be self-fixed.

¹³The magnitude is assessed using the thresholds provided by Romano et al. (2006), i.e. $|\text{delta}| < 0.147$ is negligible, $|\text{delta}| < 0.33$ is small, $|\text{delta}| < 0.474$ is medium, otherwise is large

Figure 4.1: Distribution of self-fixing rates of five TD types among 20 Python and 16 Java projects

Table 4.3: Results of Wilcoxon Rank Sum tests and Cliff's Delta Effect Size

	Code	Defect	Design	Documentation	Test
<i>p</i> -value	0.006**	0.006**	0.005**	0.034*	0.253
<i>delta</i>	-0.541 (large)	-0.544 (large)	-0.550 (large)	-0.427 (medium)	-0.252

Thus, issues related to most types of TD are more likely to be self-fixed in Python projects than in Java projects.

To investigate the self-fixing rate of TD in more depth, we consider the numerous rules for the different TD types (see Section 4.3.3). Table B.4 (in Appendix A.1) shows for each rule: the ID number, type and definition of the rule, self-fixing rate, issue fixing rate, the total number of issues and the number of projects in which violations of the rule appear. In total, Table B.4 includes 49 Python and 52 Java rules, and they are sorted in decreasing order of self-fixing rates.

Table B.4 includes 38, 41, 14, 4 and 4 rules related to Code Debt, Defect Debt, Design Debt, Documentation Debt and Test Debt, respectively. The self-fixing rates vary widely between different types of debt, i.e., from 5% to 100%. To investigate which types of debt are more likely to be self-fixed and evaluate the significance of the self-fixing rates among different types, we conducted a Scott-Knott Effect Size Difference (ESD) test (Tantithamthavorn et al., 2016) to group the five TD types into statistically distinct ranks based on the their self-fixing rates. As a variant of the Scott-Knott test (Scott and Knott, 1974), the Scott-Knott ESD test evaluates a non-normally distributed dataset and merges any two statistically distinct groups that have a negligible effect size into a single group (Tantithamthavorn et al., 2016). Table 4.4 shows the rank of the five TD types based on their self-fixing rates. We found that the five TD types are distributed in three distinct groups and, thus, the self-fixing rates are significantly different between groups. The self-fixing rates of

Table 4.4: Ranks of five debt types according to the Scott-Knott ESD tests

Group	Types of TD
1	Defect Debt, Code Debt
2	Design Debt, Documentation Debt
3	Test Debt

Defect Debt and Code Debt (group #1) are considerably higher than the other three; in other words, **Defect Debt and Code Debt receive more attention from developers who introduced them in the projects**. In contrast, Test Debt was ranked lowest (group #3) according to their self-fixing rates, indicating that **Test Debt is less likely to be self-fixed**.

4.4.3 How long does self-fixed technical debt survive during the evolution of a system?

To answer this research question, we used the Kaplan-Meier (K-M) method to analyze the survival time of the fixed issues, which is a non-parametric statistic (Kaplan and Meier, 1958). Figure 4.2 shows the Kaplan-Meier plots to visualize survival curves for all the self-fixed and non-self-fixed issues. In Figure 4.2, the left figure shows all fixed issues and their entire life cycle, i.e., the number of days between introduction and repayment. The y-axis represents the cumulative percentage of TD issues that have been fixed within a specified number of days (x-axis). In addition, the number of surviving issues is shown at the bottom of the figure. For example, there are 23,897 self-fixed issues and 45,054 non-self-fixed issues in the beginning of the projects, while 1,259 and 1,535 issues survived for more than 1,000 days, respectively.

From the left side of Figure 4.2, we found that most of both self-fixed and non-self-fixed issues were paid back in a short period of time. Since most issues have been fixed quickly, we zoomed in on the issues fixed within one month as shown on the right side of Figure 4.2. As shown in the figure, the survival curves of self-fixed and non-self-fixed issues almost overlap, and the percentage of self-fixed issues is pretty similar to that of the non-self-fixed issues.

To further investigate the survival time of different types of self-fixed issues and compare them with the non-self-fixed issues, we calculated the survival time for all issues belonging to the different types. The results are shown in Figure 4.3. Similar to Figure 4.2, the left side depicts all fixed issues in their entire life cycle for each TD type, and the right side zooms in the issues fixed within one month. A visual inspection shows that there are some differences between self-fixed and non-self-

Figure 4.2: Results of Kaplan-Meier method for the survival time of self-fixed and non-self-fixed issues

fixed issues in the 30-day window, but they do not appear to be major.

Table 4.5 shows how often an issue was fixed within a certain time-frame (i.e., one week, one month and one year) for each type of debt. To investigate whether the self-fixed issues are fixed more quickly than non-self-fixed issues, we conducted pairwise Fisher's exact tests (Fisher, 1922) on the percentages of self-fixed and non-self-fixed issues. The results confirm the aforementioned visual inspection, showing that, **for individual types of TD, there is no significant difference between the percentages of self-fixed and non-self-fixed issues within each time frame (p -values > 0.05).**

Table 4.5: Percentages of self-fixed and non-self-fixed issues within different time periods

		Code	Defect	Design	Documentation	Test
≤ 1 week	Self-fixed	73.25%	67.80%	49.09%	60.08%	76.25%
	Non-self-fixed	71.41%	60.01%	49.34%	62.99%	67.00%
≤ 1 month	Self-fixed	79.33%	72.69%	58.00%	72.34%	82.92%
	Non-self-fixed	76.20%	66.06%	55.74%	74.37%	75.03%
≤ 1 year	Self-fixed	91.78%	86.98%	80.43%	93.28%	97.08%
	Non-self-fixed	93.54%	88.02%	80.43%	96.27%	93.93%

Figure 4.3: Results of Kaplan-Meier method for the survival time for self-fixed and non-self-fixed issues belonging to different types

Table 4.6: Ranks of five debt types according to the Scott-Knott ESD tests

Group	Types of TD
1	Test Debt
2	Code Debt
3	Defect Debt, Documentation Debt
4	Design Debt

Focusing on the results related to self-fixing in Table 4.5, it seems that the issues related to Code Debt, Defect Debt and Test Debt are repaid in a shorter time. To further investigate whether the issues belonging to one TD type are self-fixed faster or slower than other types, we conducted a Scott-Knott Effect Size Difference (ESD) test (Tantithamthavorn et al., 2016) to group the five TD types into statistically distinct ranks based on their percentages for the three time-frames. Table 4.6 shows the rank of the five TD types based on their percentages of issues that have been self-fixed within the three time-frames. We found that the five TD types are distributed in four distinct groups and, thus, the percentages of self-fixed issues for all time-frames are significantly different between groups. Since the percentages of Test Debt (group #1) and Code Debt (group #2) are considerably higher than the others, it implies that **Test Debt tends to be self-fixed faster than the other four debt types, followed by Code Debt**. On the contrary, Design Debt was ranked lowest (group #4) according to their percentages, which indicates that **self-fixed issues related to Design Debt have a longer survival time**.

4.4.4 How does the number of developers and the frequency of their modifications relate with the likelihood of TD issues to be self-fixed?

This research question concerns effort sharing (i.e. the number of developers that maintain a given file) and adjusted effort sharing (i.e., the number of changes that developers made to a file). To examine these factors, we build a generalized linear mixed model (McCulloch and Neuhaus, 2005) to estimate whether and how an issue that has been self-fixed correlates with the effort sharing and adjusted effort sharing of the corresponding file in which the issue was detected. In the model, the random effect is represented by the project and file ID, and helps to account for differences between different files in terms of effort sharing and adjusted effort sharing. Moreover, we use the whole dataset (both Python and Java projects) to perform the analysis because we intend to build an explanatory model, not a predictive model. In addition, the value of the independent variables can depend on projects'

Table 4.7: Results of the generalized linear mixed model

Diagnostics					
	AIC	BIC	logLik	deviance	df resid.
	56735.7	56781.1	-28362.8	56725.7	65529
Scaled residuals					
	Min	1Q	Median	3Q	Max
	-8.5715	-0.5055	-0.1879	0.2854	9.4512
Random effects					
Groups name	Variance	Std.Dev.			
File ID (Intercept)	5.387	2.321			
Project ID (Intercept)	11.298	3.361			
Quality metrics					
Metric	OR	Estimate	Std.Error	z-value	p-value
(Intercept)	2.773	1.020	0.472	2.160	0.03
effort sharing	0.064	-2.755	0.279	-9.870	<0.001
adjust effort sharing	23.492	3.157	0.362	8.714	<0.001

characteristics, e.g., the larger a project is, the more developers might be involved in it. To properly interpret the importance of each variable in the model, we normalize variable values, within each project, in the interval $[0,1]$ ¹⁴. After that, similarly to RQ1, we build the generalized linear mixed model using the *glmer* function of the *lme4* R package (Bates et al., 2007).

Table 4.7 reports the results of the logistic regression mixed-effect model; it can be read similarly to Table 4.2. The results show that both effort sharing and adjust effort sharing have a statistically significant effect on the likelihood of an issue being self-fixed; in other words, **the number of developers involved in the files and the number of changes that developers make per file can affect the probability of the contained issues being self-fixed.**

In addition, the increase of adjusted effort sharing (i.e., based on the number of changes) can increase by about 23.5 times the odds of an issue being self-fixed. However, the increase of effort sharing (i.e., based on the number of developers) can decrease the likelihood of the issue to be self-fixed ($OR < 1$). The results indicate that **the fewer developers maintaining a file and the more changes made per file, the higher the chance of an issue being self-fixed.** More generally, we argue that when investigating effort sharing, it is not enough to look at the number of developers

¹⁴This is done by subtracting the minimum and dividing by the difference between the maximum and minimum, which was also used in Iammarino et al. (2021).

that maintain a given file, but one should also consider the amount of changes per developer.

4.4.5 Who (self) fixes technical debt?

The study involved 180 Python developers and 331 Java developers that have fixed technical debt during the evolution of the 20 Python and 16 Java projects. Among them, 96 Python and 141 Java developers have self-fixed TD. To answer this research question, we investigate the relationship between self-fixed issues and developers' *seniority*, *file count*, *commit count* and *commit size*.

The seniority refers to how long a developer has contributed to a project, calculated as the number of days between the developer's last and first commit date (see Section 4.3.4). However, if a developer has been working on a project for a long time but has contributed few commits or to few files, then this developer tends to be unfamiliar with the progress update and feel less responsible for the project. Conversely, experienced developers might be over-confident in modifying unfamiliar code, and thus inadvertently lower its quality. Therefore, it is paramount to also examine variables that reflect a developer's level of involvement in the project, i.e., the commit count and file count. We assess the two variables based on the commits performed by a developer (see Section 4.3.4). Moreover, some developers may commit fewer times or participate in fewer files, but they contribute more to some key components, which may also increase the likelihood that they will repay their own debt. Thus, we also consider the commit size, i.e., the average number of LOC that a developer contributed per commit (see Section 4.3.4).

To investigate the relationship between the mentioned variables and self-fixing, we build a GLMM where the dependent variable captures, for each TD issue, whether it is self-fixed or non-self-fixed. The random effect is represented by the project and developer ID, and helps to account for differences between developers in terms of seniority, file count, commit count and commit size. To properly interpret the importance of each variable in the model, we normalize variable values, within each project, in the interval $[0,1]$ ¹⁵.

Table 4.8 reports the results of the logistic regression mixed-effect model and is interpreted as Table 4.2. From the table, we observe that seniority and commit count have a statistically significant effect on the likelihood of an issue being self-fixed. The result indicates that **the number of days developers have been involved in a project (i.e., seniority) and the number of commits they authored (i.e., commit count) relate to whether they self-fix TD. In contrast, the number of files to**

¹⁵This is done by subtracting the minimum and dividing by the difference between the maximum and minimum, which was also used in Iammarino et al. (2021).

Table 4.8: Results of the generalized linear mixed model

Diagnostics					
	AIC	BIC	logLik	deviance	df resid.
	56113.0	56176.2	-28049.5	56099.0	61836
Scaled residuals					
	Min	1Q	Median	3Q	Max
	-8.9266	-0.6001	-0.1930	0.6890	15.9939
Random effects					
Groups name	Variance	Std.Dev.			
Developer ID (Intercept)	2.800	1.673			
Project ID (Intercept)	2.713	1.647			
Quality metrics					
Metric	OR	Estimate	Std.Error	z-value	p-value
(Intercept)	0.513	-0.667	0.406	-1.643	0.100
Seniority	0.380	-0.968	0.345	-2.807	0.005
File count	1.260	0.231	0.524	0.441	0.659
Commit count	7.849	2.060	0.406	5.071	<0.001
Commit size	1.163	0.151	0.343	0.440	0.660

which they have contributed (i.e., file count) and the average number of LOC that developers contribute per commit (i.e., commit size) have no significant effect.

In addition, the increase of commit count can increase by about 7.8 times the odds of an issue being self-fixed. However, the increase of seniority can decrease this likelihood (OR <1). These results indicate that **developers who contributed more commits are more likely to fix their debt. Furthermore, the longer the developers are involved in a project, the less often they self-fix TD.**

4.5 Discussion

In this section, we first discuss the interpretation of the findings presented in Section 4.4. Then, we describe the implications for researchers and practitioners.

4.5.1 Interpretation of Results

The Likelihood of TD to be Self-fixed

The phenomenon of self-fixed TD, as examined in this study, is widely spread in the analyzed Python and Java projects, as we observed that around half of fixed issues are self-fixed. This indicates that these developers tend to clean up their own work, paying back the technical debt they incurred themselves. However, we also found that the likelihood of TD issues to be self-fixed is negatively associated with the number of developers, the total number of issues and SLOC of projects. Thus, our results suggest that, as a project grows in size, number of developers and total issues, one can expect a decline in self-fixing rate.

To expand our understanding of the phenomenon, we delved into certain human factors related to how developers maintain the projects. In particular, we investigated the distribution of effort among components from two perspectives. First, we looked from the point of view of teams and found that, when multiple developers work in a same file, the likelihood of TD issues contained in the file and the overall project tends to decline as teams get larger. Second, we looked from the point of view of individuals and found that developers who contributed more commits are more likely to self-fix TD, while a longer involvement in the project reduces the chances of self-fixing TD. This may be an indication that a greater dedication (e.g., by providing more commits) but not seniority may reflect on a higher chance of self-fixing TD. Although we expected that commit size would render interesting observations, it displayed no significant effect; exploring the reason why is an interesting path for further research.

Putting the presented findings together, we note that even if fostering dedication among developers sounds like a good strategy, it is not necessarily desirable to push all developers to be extra active, as it would naturally lead to higher levels of collaboration on the same files, which would in turn drive the overall likelihood of TD being self-fixed down. We also note that merely driving the likelihood of TD being self-fixed up should not be a strategy, as some development strategies that serve various purposes (incl. TD remediation) can also result in lower likelihood of TD being self-fixed. Examples of such strategies include rotation of developers, contributions from external developers (e.g., via pull requests) and maintenance tactics (e.g., assigning certain types of TD items to newcomers).

Differences between Python and Java

The results of RQ2 (see Section 4.4.2) reveal that, except for Test Debt, the difference between the self-fixing rates of each type in Python and Java projects is significant.

Especially for Code Debt, Defect Debt and Design Debt, there are large effect sizes of the differences between the self-fixing rates of the two languages.

One possible reason may be related to Python's dynamic type system, which makes it popular for flexibility, expressiveness and succinctness, but also less maintainable and secure (Tratt, 2009). Developers may spend extra effort on software maintenance (Wang et al., 2015) and software quality improvement (Ray et al., 2014) (related to Code Debt and Design Debt), especially since Python code is more change-prone due to having a higher number of dynamic features (Wang et al., 2015). In addition, the misuse of dynamic features could lead to coding issues that are often fixed by adding exception handling (Defect Debt remediation) Chen et al. (2018). Altogether, Python developers may be more concerned about the issues they introduce themselves, making such TD more likely to be fixed in Python projects than in Java projects.

Types of Technical Debt and Survival Time

It is worth noting that Defect Debt and Code Debt are more likely to be self-fixed than other debt types. This may be an indication that developers are more aware of these two types or that it is easier to fix those issues. However, the Defect Debt issues seem to survive a bit longer than Code Debt issues before being self-fixed (according to the grouping results of a Scott-Knott ESD test). These observations suggest that although Defect Debt and Code Debt are preferably self-fixed and by the majority of developers (i.e., regardless of the project and team setting), developers may prioritize Code Debt.

Design Debt has quite a low self-fixing rate and this is worrisome. One study by Besker et al. (2017) shows that TD related to design has the greatest negative impact on the daily software development activities. This entails that Design Debt can be risky and that the person that may know most about the code (i.e., the developer who incurred the debt) is not addressing it sufficiently. Moreover, the situation seems worse in larger projects, where the self-fixing rate of Design Debt issues is even lower. To verify that, we calculated the Spearman correlation between the size of the project (i.e., SLOC) and self-fixing rate of Design Debt, which showed to be strongly negative¹⁶ ($\rho = -0.58$). Furthermore, we found Design Debt issues to have the longest survival time among five debt types. One way to tackle problems related to growing Design Debt, might be to focus on developers with expertise on this type of debt rather than encouraging self-fixing.

Finally, we found that developers are least likely to self-fix Test Debt. How-

---

¹⁶We interpret the correlation coefficient according to Zar (1972), i.e., a strong correlation is when $0.5 \leq |\rho| \leq 1$

ever, when developers decided to pay back their Test Debt, it was done in the shortest possible time. These results need further investigation as only four rules related to Test Debt were considered in our study (i.e. they were detected by SonarQube); other studies that focused on Test Debt analyzed ten or more test smells (Bavota et al., 2015; Palomba et al., 2016).

In summary, the results could be an indicative that developers are keen to resolve their own issues when they are related to decreasing code quality (Code Debt) or increasing chance of bugs and uncorrected known defects (Defect Debt). In contrast, developers may pay less attention to their own issues when they are related to Design Debt (e.g., long files and spaghetti code), which can affect understandability and maintainability and happen to be more often fixed by others.

Human Factors

We found that the more developers are maintaining a file, the lower the chance of an issue being self-fixed. However, the number of files developers contribute to does not appear to impact the likelihood of issues being self-fixed. Thus, strategies such as encouraging developers to take over more components (i.e., contribute to more files) may not boost TD remediation.

Furthermore, we found that developers who contribute more commits are more likely to fix their TD. This may be because these developers are more experienced or have a better understanding of the project code. Such an explanation would align with the findings by Alfayez et al. (2018) that developers with higher experience in the projects are less likely to introduce technical debt.

In a somewhat contrasting finding, Potdar and Shihab (2014) observed that developers with more experience tend to introduce more SATD. However, such a difference may be due to the nature of SATD, i.e., developers admitting debt. The more experienced developers are, the easier it is to identify TD in the source code and document it. We clarify that the results of Potdar et al. do not contradict ours since they discuss the introduction of TD (and not repayment). Furthermore, other studies showed that the experience of developers fixing SATD is higher than that of developers introducing SATD (Bavota and Russo, 2016), and that the majority of SATD is self-fixed (Maldonado et al., 2017).

4.5.2 Implications for Researchers and Practitioners

The findings and discussion drawn from this study can benefit both researchers and practitioners.

Implications for Researchers

Researchers can use the self-fixing rates to develop tools to prioritize TD remediation by assigning different weights to TD issues and giving suggestions to developers. Such suggestions could concern what types of issues to repay faster and whether they should pay the issues themselves or not. For example, TD tools could highlight critical issues that tend to survive for a longer period of time. The developer who incurred it could be notified first and, if she does not fix it, other team members could be involved.

Such tools could also monitor the source code and warn developers when an entire component is becoming a hot-spot of (potentially) long-living issues. The involved developers could be then notified and decide whether the involved TD issues should be fixed, delegated to other developers, or even ignored. Moreover, investigating how development behaviors (e.g., effort sharing) and developers' characteristics (e.g., seniority) are associated with the likelihood of the TD contained in a project being self-fixed may provide insights for predicting which projects (or components) are more likely to have unpaid TD. For example, if there is a lack of experienced developers contributing to a component, team leaders could ask more experienced developers to assess the severity of the accumulated debt. If necessary, these developers could also be allocated to test and maintain the component.

Furthermore, researchers can use our findings to design different software maintenance strategies tailored for a variety of team configurations. For example, for parts of a project that require urgent delivery, the team leader can be suggested to coordinate with experienced developers to focus on those parts. Because the experienced developers who concentrate on a certain component are better at self-fixing their TD, this practice may boost the quality of the code that has to be written in the short term (TD will be fixed quicker and better).

Implications for Practitioners

Practitioners can use the results to guide their refactoring and maintenance activities. Specifically, software development teams can use them to better assign tasks among different developers and adjust team collaboration. For example, Code Debt and Defect Debt that need to be addressed urgently can be assigned to the developer who introduced them; in contrast, Design Debt can better be assigned to other developers because it is not often self-fixed, and tends to survive longer. By knowing which TD types are more likely to be fixed in the near future, both developers and team leaders can concentrate on TD that is already overdue and has not been addressed, and on who is most suited to fix it.

Our investigation on how human factors are associated with the phenomenon

of self-fixed TD can directly support practitioners to re-allocate suitable developers to components. For example, consider a component that displays a high incidence of TD and is maintained by, among others, self-fixing-prone developers. If the accumulation of TD in the component is related to time pressure (i.e., the developers have knowledge to repay it but not enough time), a good strategy may be to let them focus their effort further on this component by reducing their workload on other components; this could effectively reduce TD through intensive self-fixing. In addition, other developers (with lower self-fixing rates) may be re-allocated to different components; this may give them an opportunity to improve their skills and knowledge of the project and become themselves self-fixing-prone developers.

Moreover, knowledge of human factors may further inform which components are more TD-prone (or less fix-prone) so that more testing effort can be budgeted for them. When maintaining those components, developers could also be encouraged to address their TD issues that belong to the debt types with a high self-fixing rate, since those issues might be more familiar and easier for them.

Finally, teams that already manage their TD by other means, can compare their current practice with our results or the results from replicating our study in their own source code, allowing for more informed decisions. For example, if a team has several open issues related to software comprehension and maintenance (e.g., long file and spaghetti code), they should be aware that the developers who introduced them are not likely to fix the issues in a short time.

4.6 Threats to Validity

In the following, we discuss the threats to construct and external validity of the reported study, as well as reliability threats.

Construct validity is related to the connection between the research questions and the objects of study. In this respect, we used SonarQube to detect TD issues in our study and, thus, our interpretation of TD is limited to the capabilities of the tool. In particular, SonarQube defines several thresholds for some rules. For example, it will flag a new violation of the rule “Lines should have sufficient coverage by tests” when a test covers less than 65% of the lines of code; and a violation of the rule “Source files should have a sufficient density of comment lines” is flagged for a file as soon as the density of comment lines on this file is less than 25%. Although SonarQube has been widely used in both industry and academia, tools using different strategies could lead to variations in the TD issues, and in turn, self-fixed issues. In principle, any used tool would be subject to similar threats.

There might also be potential threats due to false positives flagged by Sonar-

Qube. To assess SonarQube’s limitations, we manually analyzed 0.5% of the number of total fixed issues (i.e., ≈ 350) to investigate whether those issues represent technical debt and whether their evolution is accurately captured. For that, we randomly selected issues by using *stratified random sampling* (Shull et al., 2007), which is used to estimate population parameters efficiently when subpopulations have substantial variability (Cochran, 2007). For each rule, we randomly selected a number of issues based on its fixing prevalence (i.e., the results shown in Table B.4) and checked their source code to verify whether they may represent TD according to the definitions of Alves et al. (2014) and Li et al. (2015). Then, we checked if the issue was actually fixed. The results of our analysis show that all selected issues are potential TD issues (i.e., they match the used definitions). Moreover, only seven issues were not actually fixed.

There is also a threat associated with the method for filtering out issues that disappeared due to the file deletion, since it is doubtful whether developers actually aimed at fixing the technical debt. This may lead to the erroneous deletion of some issues that the developers did fix. Moreover, we cannot understand the developer’s intent by analyzing source code. Filtering out such issues is a safe way to conducting our study. In the case of file renaming, SonarQube can still track the issues in the renamed file (if the issues still exist in the system).

Another threat pertains to how we analyze the evolution of projects. Due to the amount of computational effort, we limited the data collection to 2,000 commits per project, which affected five of them. The incomplete analysis of some of the projects may have led to missing aspects of TD evolution and involved developers. This threat is partially mitigated as, the majority of issues are likely to be fixed within one year (Digkas et al., 2018), and the data collected per project spans at least one year and almost four years in average.

Moreover, although we considered four different factors (i.e., seniority, file count, commit count and commit size) for each developer, there are still other factors related to the likelihood of a developer self-fixing TD, such as the commit content and code comments. It is however, rather difficult to consider all of these factors in one study; thus we could not fully mitigate this threat.

Furthermore, mining git repositories presents potential risks. We restricted our study to the main branch only, as the investigation focuses on technical debt that has been repaid in the final product. However, different team strategies may affect how the main branch is used. The main threat in this approach is that we may be missing debt items indirectly relevant to the final product, e.g., feature branches that had debt introduced and repaid locally before merging into the main branch. Related threats also include, for example, that fixing commits might be suppressed in parent commits (i.e., squashing), or the commit might be a pull request from a

third party that is attributed wrongly. Although off-branches are out of the scope of the study, merging commits from other branches has the potential to obscure the actual author of the repayment depending on team strategies (e.g., defining who squashes commits and when). This threat can be challenging to mitigate without contacting the development team. To partially address it, we inspected a random sample of 20 (non-merge) commits from each project (approx. 1%) to check whether the commit author appears to have also authored the code (e.g., through commit comments or forked repository). This inspection suggest that the authors of the verified commits are the actual code authors.

Other related threats could be the influence of pair or mob programming practices and code review on how technical debt is managed. Such quality improvement practices sometimes are not introduced in main branches but in short-lived branches. Specifically, these practices can reduce the amount of introduced debt or increase the amount of repaid debt. We looked for evidence of such practices in the documentation and website of the projects and mainly identified code review as a common practice, although we cannot know how it affected TD management (e.g., how much extra effort was dedicated to it).

External validity concerns threats to the generalizability of our findings. The main threats stem from our study design, which considers the evolution of 20 Python and 16 Java projects from the Apache Software Foundation. Despite the credibility of the Apache Foundation and the diversity of its Python and Java projects, our results may not fully represent the entire population of non-trivial Python and Java projects. In principle, our results can be generalized to large and complex Python and Java projects in ecosystems similar to Apache. Furthermore, the set of rules considered in this study is not exhaustive. In fact, there is no complete set of issues related to TD that may affect Python or Java source code.

Reliability concerns the degree to which conducting the study depends on the involved researchers. To address these threats, three researchers were involved in the data collection and analysis. Moreover, samples of the analysis output from different steps were manually inspected for irregularities and for consistency with the proposed study design. The results showed no irregularity and all the output from different steps were consistent with the proposed study design. Furthermore, the first and second authors classified the rules independently into the five TD categories, using the description of the rules and the definition of the categories. To assess the disagreements numerically, we estimated the inter-rater agreement using Krippendorff's alpha (Krippendorff, 2011) ($\alpha = 0.74$)¹⁷. Finally, most steps were au-

¹⁷Krippendorff's inspection of the tradeoffs between statistical techniques establishes that it is customary to require $\alpha \geq 0.80$. However, where tentative conclusions are still acceptable, $\alpha \geq 0.67$ is the lowest conceivable limit (Krippendorff, 2018).

tomated by scripts which are publicly available together with the collected dataset.

Confounding factors are variables that may affect the dependent variables without the knowledge of the researchers. In our study, the main limitations that we expect pertain to the differences between Python and Java. The most prominent factors are related to the characteristics of the projects. Although language features can affect the complexity of the project, and thus affect the final dataset and findings, the complexity of the source code can also be influenced by the type of project and developers' experience, which may also not be uniform among Java and Python projects.

4.7 Conclusions

This chapter reports on an empirical study that investigated the phenomenon of self-fixed technical debt. We analyzed the evolution of TD in consecutive 17K commits of 20 Python projects and 27K commits of 16 Java projects from the Apache Software Foundation. Moreover, we detected 54K Python TD issues and 71K Java TD issues based on 49 Python rules and 52 Java rules of *blocker*, *critical* or *major* severity levels, as defined by SonarQube.

We found that about half of fixed TD issues were repaid by the same developers who introduced them. Moreover, the projects that are larger and have more commits and technical debt issues tend to have a relatively lower likelihood of TD issues being self-fixed. We also observed that there is no significant difference in the survival time between self-fixed and non-self-fixed issues.

Looking at the types of TD, our findings suggest that Defect Debt and Code Debt are more likely to be self-fixed than other debt types. The results indicate that developers mostly pay back technical debt that was introduced by them when it is related to lower code level issues (e.g., compared to Design Debt and Documentation Debt).

Finally, regarding developer-related factors, the results indicate that those who are more dedicated to or knowledgeable of the project (e.g., by contributing more commits) may drive the overall self-fixing up. Moreover, the longer the developers are involved in the project, the less often they self-fix TD.

This chapter reported the first study that exclusively focused on the phenomenon of self-fixed technical debt, and looked at two of the most popular programming languages, i.e., Python and Java. However, the related findings are based on source code analysis and not on the opinion of software developers. Thus, to more broadly comprehend the phenomenon of self-fixed TD, the next chapter reports on investigating the phenomenon of self-fixed TD from the perspective of practitioners.

Based on:

Jie Tan, Daniel Feitosa, Paris Avgeriou, (2021) “Do practitioners intentionally self-fix Technical Debt and why?” in Proceedings of the 37th IEEE International Conference on Software Maintenance and Evolution (ICSME ’21), Luxembourg City, DOI=10.1109/ICSME52107.2021.00029

Chapter 5

Do practitioners intentionally self-fix Technical Debt and why?

Abstract

The impact of Technical Debt (TD) on software maintenance and evolution is of great concern, but recent evidence shows that a considerable amount of TD is fixed by the same developers who introduced it; this is termed *self-fixed TD*. This characteristic of TD management can potentially impact team dynamics and practices in managing TD. However, the initial evidence is based on low-level source code analysis; this casts some doubt whether practitioners repay their own debt intentionally and under what circumstances. To address this gap, we conducted an online survey on 17 well-known Java and Python open-source software communities to investigate practitioners’ intent and rationale for self-fixing technical debt. We also investigate the relationship between human-related factors (e.g., experience) and self-fixing. The results, derived from the responses of 181 participants, show that a majority addresses their own debt consciously and often. Moreover, those with a higher level of involvement (e.g., more experience in the project and number of contributions) tend to be more concerned about self-fixing TD. We also learned that the sense of responsibility is a common self-fixing driver and that decisions to fix TD are not superficial but consider balancing costs and benefits, among other factors. The findings in this chapter can lead to improving TD prevention and management strategies.

5.1 Introduction

The term *technical debt* (TD) is commonly used to express the adoption of suboptimal solutions in software development, that impact maintainability and evolvability (Li et al., 2015; Kruchten et al., 2012). To help practitioners in managing TD, researchers have investigated potential causes and effects of TD (Rios et al., 2020; Behutiye et al., 2017; Besker, Martini, Lokuge, Blincoe and Bosch, 2018), its various

types (e.g., Code (Palomba et al., 2018b; Di Nucci et al., 2018), Design (Abidi et al., 2021; Sharma et al., 2020), and Test Debt (Spadini et al., 2018; De Bleser et al., 2019)) and the role of tools (Avgeriou et al., 2020; Lenarduzzi et al., 2021). The community has also invested effort into studying specific forms of TD such as *self-admitted TD*, i.e., when developers acknowledge introducing TD themselves (e.g., in source code comments) (Maldonado et al., 2017; Iammarino et al., 2019). More recently, there is work on *self-fixed TD*, i.e., when developers repay TD that they introduced themselves (Tan et al., 2020).

Although investigating self-fixed TD is still rather preliminary, initial findings are noteworthy. For example, Tan et al. (2020) found that more than two-thirds of TD items in source code are self-fixed, and that Defect Debt tends to be self-fixed frequently, while Test Debt and Design Debt are likely to be repaid by other developers (Tan et al., 2020). However, current research on this topic, similarly to most work on TD, is based on source code analysis and not on the opinion of software developers. Consequently, there is no evidence of whether developers consider the differences between various debt types when repaying their own debt and what else they take into account. We advocate that, in order to comprehend the phenomenon of self-fixed TD more thoroughly, it is paramount to consider the practitioners' intent and experiences.

To address this shortcoming, this chapter reports on a study aiming at examining the phenomenon of self-fixed TD from the perspective of practitioners. To this end, we conducted a survey with 17 well-known Java and Python open-source software communities, receiving 181 valid responses (89 for Java and 92 for Python). The survey included questions on the practitioners' attitude towards introducing and repaying TD in general and regarding five types of debt: Code Debt, Defect Debt, Design Debt, Documentation Debt and Test Debt. These five types have been extensively investigated in other studies (Baldassarre et al., 2020; Digkas et al., 2018; Bavota and Russo, 2016), and were also the focus of a related work on self-fixed TD (Tan et al., 2020). The survey also looks into practitioners' reasons to introduce and self-fix TD, as well as human factors that may influence their responses (e.g., experience and contribution levels).

This study offers a number of contributions. First, it provides insights on the relevance of self-fixed TD to practitioners and the recurrence with which they address their own debt. Second, it elaborates on the circumstances that lead to self-fixing, also providing a list of reasons (both technical and non-technical) and highlighting how these associate with reasons to introduce TD. These contributions can benefit both researchers and practitioners. On the one hand, researchers can build on this evidence to fine-tune TD-related tools and further study the phenomenon of self-fixed TD. On the other hand, practitioners can gain a broader understanding of the

intent for introducing and self-fixing TD, which can help them devise or refine TD prevention and management strategies.

The remaining sections are organized as follows. The chapter starts by summarizing related work and contextualizing our study in Section 5.2. Next, Section 5.3 reports on the goal, research questions and methodology of the study. Then, the results are presented in Section 5.4, while Section 5.5 elaborates on their interpretation and implications to researchers and practitioners. Section 5.6 reports on the threats to the validity of our study, leading to Section 5.7 where we conclude the chapter.

5.2 Related Work

Recent studies have shown that a large percentage of TD is paid back during software evolution (Tan et al., 2021; Digkas et al., 2018; Lenarduzzi et al., 2019; Baldassarre et al., 2020). Tan et al. (2020), established that a considerable amount of TD items are repaid by the same developers who introduced them. More recently, a number of studies focused on self-admitted technical debt (SATD): TD items that are introduced intentionally and explicitly documented in code comments (Potdar and Shihab, 2014; Bavota and Russo, 2016; Fucci et al., 2020).

Some related findings indicate that the majority of SATD is removed by the same developer who introduced it (Maldonado et al., 2017; Bavota and Russo, 2016; Liu et al., 2021). However, all these previous studies drew conclusions only through source code analysis (including source code (Baldassarre et al., 2020) and code comments (Liu et al., 2021)) and focused only on one language (i.e., either Python (Tan et al., 2020) or Java (Maldonado et al., 2017)). In our study, we conducted a survey to investigate the practitioners' perspective on how they self-fix TD for both Java and Python.

Other studies have focused on the rationale in TD management, e.g., the reasons to introduce or (not) repay TD. For example, Rios et al. (2020) reported that non-technical reasons, e.g., *deadline* and *inappropriate planning*, play a significant role in the occurrence of TD items. Some studies also pointed out that the leading causes of technical debt are architectural choices (Ernst et al., 2015) and that developers are frequently forced to introduce new TD due to already existing TD (Besker, Martini and Bosch, 2018). In addition, Maldonado et al. (2017) found that developers self-admit TD to track potential future bugs, code that needs improvements or areas to implement new features, while they mostly remove SATD when they are fixing bugs or adding new features. Some SATD may also be accidentally removed when entire classes or methods are dropped (Zampetti et al., 2018). Researchers have also investigated the reasons for not paying TD back, e.g., *lack of organizational interest*,

low priority on the debt, focus on short-term goals, cost, and lack of time (Freire et al., 2020). In contrast to these studies, we focus on the reasons influencing developers' decisions to deal with their own debt, i.e., why they introduce and self-fix TD.

There is also work on the influence of human factors on incurring and managing TD. Some findings indicate that most developers that incur TD have low project-related experience (Amanatidis et al., 2017) while developers' seniority and commit frequency are negatively correlated with the amount of introduced TD (Alfayez et al., 2018). In addition, Besker, Martini, Lokuge, Blincoe and Bosch (2018) found that some organizational factors, such as developers' experience and startup founders' knowledge, can influence the intentional accumulation of TD. However, another study found that the developers' participation level (i.e., the total lines of code edited by a developer) and their experience in the project positively correlate with the amount of TD introduced by them, while communication skills have barely any impact on TD (Salamea and Farré, 2019). In contrast to these studies, we focused on remediation of TD rather than its introduction, and in the context of self-fixed TD.

5.3 Research approach

The goal of our study, described according to the Goal-Question-Metrics (GQM) approach (Van Solingen et al., 2002), is to “*analyze practitioners' attitudes for the purpose of investigating self-fixed TD with respect to the intent and rationale for remediating it and the influence of human factors in doing so from the point of view of software practitioners in the context of open-source software development*”. In the following subsections, we describe the research questions, the survey design, participants and projects selection, as well as data analysis in detail.

5.3.1 Research Questions

We further refine the goal of this study into the following research questions:

RQ₁: Do practitioners intentionally self-fix TD?

This research question aims at investigating to what extent developers repay their own debt intentionally. More specifically, we analyze practitioners' opinions to examine which types of debt they self-fix intentionally. The results can shed further light on how developers prioritize the remediation of technical debt. For example, developers may introduce specific types of debt with the intention of repaying it soon thereafter, thus prioritizing this item above others.

RQ₂: What reasons motivate practitioners to introduce and self-fix TD?

This research question aims at identifying practitioners' rationale for introducing and subsequently repaying their own debt. Analyzing the possible reasons can

help to manage debt (in the case that TD is unavoidable) or prevent it (if possible). An example of management is to build tools that account for and highlight the conditions under which developers are more likely to repay a debt item. An example of prevention is to identify those reasons for introducing debt, that can be eliminated depending on a particular team's culture and composition, e.g., by making a more fine-grained planning or avoiding overloading the team.

RQ₃: How do human factors influence practitioners in self-fixing TD?

Practitioners in different roles, with different experience and contribution levels may have different attitudes towards self-fixing TD. For example, developers with a leading role tend to monitor and manage the development tasks more often than others (Oliveira et al., 2020; Trendowicz and Münch, 2009), which may reflect in a stronger attitude towards repaying their own debt. Also, practitioners with more experience are more likely to repay technical debt (Bavota and Russo, 2016). Thus, investigating the relationship between characteristics of practitioners and how they self-fix TD can help comprehend and subsequently support their decision-making processes when managing TD. For example, team composition can be adjusted to foster TD remediation by taking into account the diversity of roles, experiences, and cultures. In addition, researchers can also use the results to offer different tool options for practitioners with different characteristics.

5.3.2 Survey Design

According to our goal and the guidelines for selecting empirical methods in software engineering research (Easterbrook et al., 2008), we decided to use a survey. A survey is a comprehensive research method for describing, comparing, or explaining knowledge and behavior (Fink, 2003); it thus suits our focus on collecting and explaining practitioners' perceptions on self-fixed TD. More specifically, we decided to use a web-based questionnaire to conduct the survey because it is time- and cost-effective, and also suitable for collecting data from a large number of developers in geographically diverse locations (Lethbridge et al., 2005).

The flow of the survey design is presented in Figure 5.1. The black box represents the introductory welcome page, and the numbered white boxes depict steps in which participants answer questions. The data collection was planned to last two months and the questionnaire was made accessible online between February and March 2021. To mitigate privacy concerns, all invited participants were informed about the survey with an invitation text describing the target open-source project and programming language (see Section 5.3.3), together with the motivation and goal of this study. Moreover, we did not collect any identifiable information in the survey. The templates used for communication are available in the replication pack-

age¹.

Figure 5.1: Design of the survey in our study.

As shown in Figure 5.1, the survey started with a welcome page explaining the structure of the questionnaire, and that the survey is anonymous. On step ①, the survey starts with a form with demographic questions. In particular, we inquired about the participants' role in the target software project (e.g., developer, tester), their educational background, the number of years they have been working in software development, the number of years they have been working on the project, and the number of commits they have contributed to the project. Moreover, since this survey targeted a broad audience, we wanted to assess how balanced the population is and, thus, we also asked for the participants' sex and country (as optional questions).

Subsequently, the survey included five general questions (see Table 5.1) related to technical debt in step ②. But before asking any questions, we start by describing three terms to set a baseline for common understanding: what *technical debt* is, and what it means to *introduce* it and *repay* it. We then ask GQ1 to identify whether the participants are familiar with the aforementioned descriptions of these terms. Next, to investigate whether practitioners are aware of introducing and repaying their TD, we record the frequency (from *always* to *never*), with which practitioners introduced (GQ2) and self-fixed (GQ4) TD. We also ask participants the reasons why they have introduced (GQ3) and self-fixed (GQ5) TD. The options for GQ3 stem from the results of a previous study (Rios et al., 2018), which reported the top ten most cited TD causes as informed by 107 participants. GQ5 is an open-ended question as there was no established knowledge in the literature on reasons for self-fixing TD. Finally, for GQ3, in addition to the listed options, we add the option *Other* with a text field for participants to provide more details.

The questions in step ③ covers all five types of debt. For each debt type, we provide the definition and ask the participants about their recollection of (DQ1) *noticing* and (DQ2) *self-fixing* TD items. For that, we use a Likert scale with the options *definitely not*, *probably not*, *I am not sure*, *probably yes*, and *definitely yes*.

¹<https://github.com/sftdsoftware/icsme>

Table 5.1: TD general questions of the survey

No.	TD General Question Description
GQ1	Were you familiar with the presented description of technical debt before this questionnaire?
GQ2	How often do you introduce technical debt?
GQ3	For what reasons have you introduced technical debt?
GQ4	How often do you repay your own technical debt?
GQ5	For what reasons do you repay your own technical debt?

5.3.3 Projects and Participants Selection

For this study, we selected projects that are written in either Java or Python, which are ranked as the top two most popular object-oriented programming languages². Although both of them are strongly typed languages, Java is statically typed while Python is dynamically typed. Because of the dynamic features, Python projects are more change-prone, resulting in developers spending more effort on software maintenance (Wang et al., 2015) and software quality improvement (Ray et al., 2014). Thus, selecting systems written in these two programming languages, helps to increase the external validity of our study. Moreover, investigating both languages can help to compare the findings to the previous results that focus on source code analysis (Digkas et al., 2018; Tan et al., 2020). In the following, we describe the process of selecting projects and participants for the survey.

We started by compiling a list of systems that have been widely used in the studies that investigated technical debt through source code analysis (see Section 5.2). From this list, we filtered out projects with less than 5,000 commits or 200 contributors, to ensure that a sufficient number of practitioners are familiar with the maintenance of the projects for this survey. Ultimately, the following eleven projects were selected: CAMEL (Maldonado et al., 2017; Iammarino et al., 2019), CASSANDRA (Wehaibi et al., 2016), CHROMIUM (Wehaibi et al., 2016; Bellomo et al., 2016), ELASTIC-SEARCH (Flisar and Podgorelec, 2018), GERRIT (Maldonado et al., 2017; Iammarino et al., 2019), HADOOP (Yan et al., 2018; Zampetti et al., 2020), HBASE (Palomba et al., 2018a), GUAVA (Lerina and Nardi, 2019; Tsoukalas et al., 2020), JENKINS (Tsoukalas et al., 2020; Lerina and Nardi, 2019), RXJAVA (Lerina and Nardi, 2019; Iammarino et al., 2019), TENSORFLOW (Liu et al., 2021). Considering that, except for TENSORFLOW, these are Java projects, we still needed to supplement the list with pertinent

²According to the Tiobe Index, which is one of the best-known indexes of programming languages popularity: <https://www.tiobe.com/tiobe-index/>, visited March 2021

Table 5.2: Mapping between survey questions, variables and research questions

No.	Variables	RQs
BQ1 – BQ7	demographic information, i.e., the practitioners' role, educational background, sex, country, contribution, experience in the project and software development	RQ ₃
GQ1	practitioner's familiarity with technical debt	RQ ₁
GQ2	the frequency of introducing technical debt	RQ ₂
GQ3	the reasons why developers have introduced TD	RQ ₂
GQ4	the frequency of self-fixing technical debt	RQ _{2,3}
GQ5	the reasons why practitioners have self-fixed TD	RQ ₂
DQ1	the types of debt that practitioners have noticed	RQ ₁
DQ2	the types of debt that practitioners have self-fixed	RQ _{1,3}

Python projects. In addition to complying with the same thresholds for the number of commits and contributions, we chose the Python projects with the most stars on GitHub: ANSIBLE, DJANGO, FLASK, PANDAS, SCIKIT-LEARN, and SUPERSET. The number of stars of a GitHub repository can be seen as a proxy of its popularity (Borges et al., 2016) as more stars indicate that more users are interested in the project (Tsay et al., 2014).

To obtain the target sample of participants, we extracted each contributor's name, email address, and the time of each commit from GitHub. Next, for each project, we consider the contributors who had submitted at least two commits within the last three years; this helps to target contributors who still have a recollection of their work in the project and whose email address is likely to still be in use, thus improving the response rate and the response quality. To mitigate privacy concerns, we did not retain any information after sending invitations. In total, we sent out 7638 emails.

5.3.4 Data Analysis

Table 5.2 presents the mapping between the questions in the survey, the collected variables and research questions. The three sets of survey questions correspond to the three steps in the survey design (i.e., step ① – ③ in Figure 5.1): background (demographic) questions (BQ), general questions about technical debt (GQ), and detailed questions about TD types (DQ). In the following, we describe in detail how each of the research questions makes use of the responses.

To answer RQ₁, we first investigate to what extent developers are familiar with the description of technical debt from the responses of GQ1, using descriptive statis-

tics. Next, we answer to what extent developers self-fix technical debt intentionally per TD type; for that, we use the responses of DQ2 from step ③ of the survey (i.e., *Do you recall self-fixing technical debt (per type)?*). Since the responses to this question use a five-point Likert scale (i.e., from *definitely not* to *definitely yes*—see Section 5.3.2), the results are presented in the form of stacked bar charts and comparisons are assessed using statistical tests. We also make a supplementary discussion in conjunction with the types of debt that participants have noticed in source code (DQ1), since identifying TD items may contribute to one’s knowledge of TD.

To answer RQ₂, i.e., the reasons why practitioners introduce and self-fix TD, we first investigate the extent to which they are aware of introducing and self-fixing debt (GQ2 and GQ4, respectively). For that, we create and analyze a heat map comparing various frequencies of TD introduction to various frequencies of self-fixing it (see Section 5.3.2) and evaluate the association between them using several statistical tests. Then, we qualitatively discuss the reasons provided by participants to explain why they decide to introduce (GQ3) and self-fix technical debt (GQ5). Since GQ3 provides the option *Other* (with a text field) and GQ5 is an open-ended question, we followed an open-coding approach (Seaman, 1999; Miles and Huberman, 1994) to analyze the answers collected from the survey respondents; a similar approach was followed in many recent software engineering studies, e.g., (Aghajani et al., 2020; Vadlamani and Baysal, 2020). To conduct open coding, the first two authors independently assigned one or more tags to each answer; each tag is meant to summarize a reason. The level of inter-rater agreement between the classifications of the two authors was measured and conflicts were solved by a discussion with the third author. Since participants can provide multiple reasons to introduce and self-fix TD, the associations between reasons are presented in the form of a chord diagram and investigated using descriptive statistics.

For RQ₃, to explore how the human factors (characteristics of the practitioners such as roles, experience and contribution) affect their self-fixing behavior, we investigate the relationship between the demographic information of practitioners (step ① in Figure 5.1) and their perspective of self-fixing TD (step ②③ in Figure 5.1). For that, we first calculate, for each demographic group, the proportions of different ratings for the frequency (GQ4) and certainty (DQ2) of self-fixing TD. Then, we conducted several statistical tests to explore whether one demographic group tends to rate the frequency or certainty higher or lower than other groups.

Table 5.3: Characteristics of the respondents

Characteristics	Java	Python	Characteristics	Java	Python
Main role in the project			Experience in the project		
Developer*	76.40%	69.56%	< 1 year	8.99%	33.70%
Software Architect	8.99%	14.13%	1–2 years	26.97%	36.96%
Team Leader	6.74%	4.35%	3–5 years	32.58%	15.22%
Product Owner	3.37%	5.43%	> 5 years	31.46%	14.13%
Tester**	1.12%	2.17%	Contribution to the project		
System Administrator	1.12%	1.09%	< 10 commits	11.24%	43.48%
DevOps Engineer	1.12%	1.09%	10–50 commits	12.36%	14.13%
Engineering Manager	1.12%		50–100 commits	10.11%	14.13%
Project Manager		2.17%	100–200 commits	12.36%	6.52%
Educational background			200–500 commits	14.61%	7.61%
Master degree	48.31%	30.43%	> 500 commits	39.33%	14.13%
Bachelor degree	39.33%	36.96%	Software development experience		
Ph.D. degree	8.99%	20.65%	< 5 year	12.36%	21.74%
High school	1.12%	4.35%	5–10 years	16.85%	26.09%
Other / not shared	2.25%	7.61%	10–20 years	44.94%	28.26%
Geographical location			20–35 years	23.60%	18.48%
Europe	52.81%	29.35%	> 35 years	1.12%	4.35%
North America	21.35%	39.13%	Not shared	1.12%	1.09%
Asia	10.11%	23.91%	Respondent's sex		
South America	5.62%	2.17%	Male	87.64%	90.22%
Oceania	2.25%	1.09%	Female	3.37%	6.52%
Not shared	7.87%	4.35%	Other	1.12%	
			Not answered	7.87%	3.26%
*Developer, incl. Software Engineer			**Tester, incl. Quality Assurance		

5.4 Results

Overall, we obtained 184 responses to our survey and the response rate is 2.41%. Among them, 181 responses were valid: 89 for Java and 92 for Python. The three invalid responses lack specific information as, e.g., respondents choose “Other” and answered “NA” for several questions. The distribution of the population is presented in Table 5.3. In the following, we present the results of the study organized by research question.

5.4.1 Do practitioners intentionally self-fix technical debt? (RQ₁)

To answer this research question, we first investigated to what extent developers are familiar with the concept of technical debt (GQ1 in Table 5.1). Figure 5.2 depicts the answers through a diverging stacked bar chart to visualize the percentages that describe the distribution of responses. The left side shows the answers of the 89 Java participants (49%), while the answers of the 92 Python participants (51%) are presented on the right.

Figure 5.2: Respondents' knowledge level of technical debt

According to Figure 5.2, the vast majority (i.e., 92%) of the participants were familiar or somewhat familiar with technical debt before this survey. This increases our confidence that the data we collect is (mostly) derived from knowledgeable individuals (a thorough discussion of threats to validity is found in Section 5.6). In addition, Java participants seem slightly more familiar than Python participants. To further evaluate the significance of such difference, we calculated the Wilcoxon Rank Sum test (Armstrong, 2014) (p -value = 0.02) and Cliff's Delta Effect Size (Cliff, 2014) (δ = 0.1468) on the ratings of the Java and Python respondents, since we cannot assume the populations are normally distributed. The results reveal that the difference between the familiarity of Java and Python participants with TD is significant (i.e., p -value < 0.05) but the effect size is negligible (i.e., $|\delta|$ < 0.147). Thus, different levels of familiarity with the concept of TD between Java and Python participants have almost no effect on the disparity between the populations in subsequent findings.

Delving into the essence of the research question, we looked at whether participants recalled self-fixing each type of debt (DQ2). While incurring TD can be both intentional and unintentional, self-fixing in this case is intentional: if practitioners recall paying back their own TD items, it is safe to assume they did it on purpose (at least for those items). The answers are summarized in Figure 5.3. This figure shows that, for each type of TD, more than half of the participants mentioned that they *definitely* or *probably* have self-fixed items. To investigate the significance of this observation, we first combined the answers into three categories: (1) positive (*definitely* or *probably* self-fixed TD); (2) neutral (*I am not sure*); and (3) negative (*definitely not* or *probably not* self-fixed TD). Next, we conducted a Scott-Knott Effect Size Difference

Figure 5.3: Distribution of self-fixed TD types

(ESD) test (Tantithamthavorn et al., 2016) to group the three categories of answers into statistically distinct ranks based on the combined proportions of all debt types. As a variant of the Scott-Knott test (Scott and Knott, 1974), the Scott-Knott ESD test evaluates a non-normally distributed dataset and merges any two statistically distinct groups that have a negligible effect size into a single group (Tantithamthavorn et al., 2016). The result shows that the three categories belong to different groups and, thus, the percentage differences observed are significant. Since the percentage of positive answers (category #1) is considerably greater than the other two, it implies that **developers often display a positive attitude towards self-fixing TD**.

Looking into individual deb types, participants seem to be slightly more concerned about self-fixing Code Debt issues, less about Test Debt, and mostly uncertain about Design Debt. To assess the significance of these differences, we conducted a Scott-Knott ESD test to group TD types into statistically distinct ranks based on the Likert scale. The result shows that with the exception of Code Debt, there is no significant distinction between debt types. Consequently, **as Code Debt displays the highest positive score, it seems to be the main concern of the participants**.

Comparing the languages, we observed from Figure 5.3 that Java participants seem to be more likely to acknowledge that they have *definitely* or *probably* self-fixed TD, as more than half of them mentioned so for each debt type. To assess this particular observation, we conducted two Scott-Knott ESD tests (for Java and Python populations respectively) to group TD types into statistically distinct ranks based on their Likert scores. The result shows that there are three types of TD (i.e., Code Debt, Test Debt, and Defect Debt) belonging to the same group in Java, while Code Debt alone makes up one group in Python. **Although Code Debt is reaffirmed as an important concern in both languages, Test Debt and Defect Debt are as relevant as Code Debt only to the Java participants**.

5.4.2 What reasons motivate practitioners to introduce and self-fix TD? (RQ₂)

To answer this research question, we first investigated the extent to which practitioners are introducing and self-fixing TD. Figure 5.4 shows a heat map depicting the percentages of participants who self-fixed TD at different frequencies for a given frequency of TD introduction in Java and Python, respectively. A Wilcoxon Signed Rank test (Rosner et al., 2006) reveals no significant difference between Java and Python (p -value = 0.29), suggesting that Java and Python participants introduce and self-fix TD in a similar fashion.

Figure 5.4: Percentage of participants indicating the frequency of self-fixing TD (column) for a given frequency of TD introduction (row).

Analyzing the population as a whole, almost half of the participants (48%) have acknowledged that they self-fix TD *on a regular basis* and a half of those (27% of the general population) acknowledge also introducing debt sometimes only. This combination seems more recurrent than any other. To assess the significance of this observation, we conducted a Scott-Knott ESD test to group the frequencies of self-fixing into statistically distinct ranks based on introduction frequencies. The result shows participants that self-fix TD *on a regular basis* belong in a distinct group, and participants that do it *sometimes* in another distinct group. The results indicate that **although participants sometimes have to introduce TD, when they do, they also tend to pay it back.**

In the following, we examine the reasons that lead practitioners to introduce and self-fix TD. Regarding the former, 37 of all the 181 participants also selected the

Figure 5.5: Reasons to introduce technical debt

option *Other* and provided custom opinions. We applied an open-coding approach and extracted one or more reasons from each custom answer that could either fit into one of the ten pre-defined reasons or become a new reason. By the end of the coding, five new reasons had been created. Figure 5.5 shows the 15 reasons that can lead to TD introduction from the participants' perspective. The overall frequency of each reason is the number of participants who mentioned it. Apart from the reasons listed in the Figure, 20 participants selected that they *are not aware of introducing TD*. Another four participants indicated that *TD introduction is inevitable*, e.g., "all code is technical debt" and "TD introduction is a part of software development"; one of them mentioned also the reason *non-adoption of good practice*.

To facilitate the interpretation of the reasons to incur TD, we grouped them into more general categories, which we adopted from Rios et al. (2020). The categories are also shown in Figure 5.5, from which we observe that *deadline* is cited by more than half of the participants, indicating that it is a factor that normally contributes to the appearance of TD. During the open-coding, some custom answers were coded as *deadline*, e.g., "it is evaluated and decided to be worth incurring this technical debt to finish the task earlier". Furthermore, *lack of experience*, *inappropriate planning*, and *team overload* are other reasons cited by at least 30% of the participants. Among them, *lack of experience* also includes custom answers such as "lack of learning" and "not fully understanding how the project will evolve over time".

The five new reasons that appeared during the open-coding are: *workload balance*

referring to the trade-off between the cost and benefit of introducing TD, e.g., “prioritize mainline tasks and de-prioritize secondary tasks”, and “increasing tech debt is desirable”; *not well-understood technology or feature*, e.g., “experiments to introduce new technologies” and “a feature might not be well understood and it is more cost effective to do it in a suboptimal way”; *debt inheritance* referring to new debt introduced due to a pre-existing TD item, e.g., “working in an area that is already burdened with technical debt”; *backward compatibility*, e.g., “backward compact and minor version release rules” and “backportability”; and *lack of refactoring*, e.g., “ongoing major refactoring”.

Regarding the reasons to self-fix TD, we summarized 25 reasons from the responses (all open-ended in this case), as shown in Figure 5.6. Similarly to Figure 5.5, we also mapped the reasons into more general categories. For that, we adopted the categories provided by Rios et al. (2020) again, but renamed some categories to fit the context. During the open-coding, answers were often mapped into multiple reasons. For example, one participant answered: “improving code quality, maintainability, making new features easier to implement”. This answer was coded as *improve quality*, *improve maintainability*, and *prepare for new features*.

After examining the reasons for self-fixing TD, we noticed that *sense of responsibility* and *avoid higher cost* are the most frequently mentioned, each being cited by at least 37 participants (approx. 20%). For example, one participant mentioned “I feel ownership over the code, also, better to get rid of it before it bites you”, which was coded as both *sense of responsibility* and *avoid higher cost*.

Most participants mentioned multiple reasons when answering why they introduce and self-fix TD. Thus, to gain a better understanding on the relationship between these reasons, we calculated the number of times that every pair of categories was mentioned together (i.e., totally 1536 relations), and plotted a chord diagram (Gu et al., 2014) illustrating the relations between categories of reasons (cf. Figure 5.7). The categories for introducing and self-fixing TD are colored red and blue, respectively, and the width of the links conveys the strength of the relation.

From Figure 5.7, we can observe that among all the relations, the reasons to introduce TD (in red) seem to display the strongest relationships with themselves. In particular, the reasons related to *planning and management* and *methodology* were most often mentioned simultaneously (127 times), followed by *planning and management* and *people* (121 times). Regarding the reasons for self-fixing TD, it is noticeable that the links between the self-fixing and introduction categories are wider than those between self-fixing reasons themselves. In other words, the reasons for self-fixing TD were mentioned more often together with those to introduce TD than with themselves (629 vs. 112 times). Especially the reasons to self-fix TD related to *development* were often mentioned when the reasons to introduce TD concerned *planning*

Figure 5.6: Reasons to self-fix technical debt

and management (108 times).

Altogether, we observe both proactive reasons (e.g., *sense of responsibility*, *avoid higher cost*) and reactive reasons (e.g., *TD overload*, *Repayment request*) to self-fix TD. More importantly, the evidence suggests that **regardless of the reason that lead a practitioner to introduce TD, she is keen to repay the debt proactively** not only due to an internal driver (e.g., *sense of responsibility* or *improve collaboration*) but also for strategic reasons (e.g., *avoid higher debt* or *improve maintainability*). Further evidence that repayment is thoughtfully considered, is that most reasons to self-fix TD co-occur with reasons to introduce TD related to *planning and management*.

5.4.3 How do human factors influence practitioners in self-fixing TD? (RQ₃)

To answer this research question, we followed a process similar to that used by Wan et al. (2018) and Zou et al. (2018) to analyze 11 demographic groups derived from the *team roles*, *development experience*, *project experience* and *contribution to project*. For

Figure 5.7: Association between reasons

comparison reasons, we also consider an additional group (All) including all participants. We explain the demographic groups in the following paragraphs.

Regarding the *team roles*, one group corresponds to developers (Dev), including software engineers. Another group consists of respondents who are involved in the project or team management and support (PM), i.e., software architect, team leader, product owner, system administrator, DevOps engineer, engineering manager and project manager. In addition to these two groups, we also received three responses from testers. However, we did not consider testers as a separate group because the sample size does not fit the threshold for statistical analysis (Siegel and Castellan, 1988), which would also increase the randomness of the results (Lenth, 2001).

Regarding the levels of *software development experience*, we distinguish between respondents with: (a) high experience (ExpSdHigh), as the 25% with the most experience in years (Zou et al., 2018), i.e., ≥ 18 years in this survey; low experience (ExpSdLow), as the 25% with the least experience in years, i.e., ≤ 5 years; and medium experience (ExpSdMed), i.e., the remaining respondents. For *project experience*, since the survey offered discrete options, we selected the maximum and mini-

Figure 5.8: The percentages of how often participants in each demographic group have self-fixed TD

mum interval that approximates the top and bottom 25%. In particular, we defined a high experience group (`ExpProHigh`) with those that have five or more years in a project (approx. 23%), a low experience group (`ExpProLow`) with those that joined a project less than one year ago (approx. 22%), and a medium experience group (`ExpProMed`) with the remaining respondents. The *contributions* to projects are grouped in a similar fashion, as follows: high contribution level (`ContribHigh`), with participants that submitted 500 commits or more (approx. 27%); low contribution level (`ContribLow`), with those that submitted up to 10 commits (approx. 28%); and medium contribution level (`ContribMed`), with the remaining respondents.

Figure 5.8 presents the frequencies of self-fixing TD as perceived by respondents in the eleven demographic groups. As shown in the figure, the majority of participants in all of the 12 groups (79%–97%) mentioned they repay their TD *always*, *on a regular basis*, or *sometimes (when it is absolutely essential)*. In particular, about half of the participants in the various groups (44%–54%) considered they self-fix TD *on a regular basis*.

Figure 5.9 depicts the certainty of respondents in each demographic group about having self-fixed all five TD types (combined); this is in contrast to Figure 5.8 which reflects frequency, independently of TD types. For example, in the group `All`, out of a total of 905 ratings, 36.5%, 21.9%, 15.0%, 11.4%, and 15.2% respectively were *definitely yes*, *probably yes*, *I am not sure*, *probably not*, and *definitely yes*. As shown in the figure, more than half of the participants in most groups mentioned that they have self-fixed TD (43%–80%). However, these percentages are slightly lower than the percentages of the participants who mentioned that they have self-fixed TD *always*, *on a regular basis*, or *sometimes* (i.e., 79%–97% in Figure 5.8). This discrepancy

Figure 5.9: The percentages of self-fixing certainty levels for participants in each demographic group

is due to the fact that we focused on individual types of TD when asking about the certainty level; thus, the participants might only self-fix specific types.

To investigate whether one demographic group tends to self-fix TD more or less frequently than other groups, we conducted pairwise Fisher's exact tests (Fisher, 1922) with Bonferroni-Holm corrections (Aickin and Gensler, 1996) on the proportions of respondents from the different demographic groups. In detail, those statistical tests were conducted within four sets of demographic groups, i.e., a set of different roles (Dev versus PM), a set of different levels of software development experience (ExpSdHigh versus ExpSdMed versus ExpSdLow), a set of different contribution levels (ContribHigh versus ContribMed versus ContribLow) and a set of different levels of project experience (ExpProHigh versus ExpProMed versus ExpProLow). Moreover, for the demographic groups in each group set, we conducted eight times pairwise Fisher's exact tests with Bonferroni-Holm corrections to validate the significance of rating differences among these groups. They are the tests on the proportions of different frequencies ratings, together with the attitudes towards self-fixing TD (i.e., *always*, *on a regular basis*, and *sometimes, when it is absolutely essential*) and the negative ones (i.e., *almost never* and *never*). Furthermore, similar to RQ₁, we also conducted the tests twice: one to compare *definitely yes* and *probably yes* between the demographic groups; and another to compare *definitely not* and *probably not*. The results per set, are as follows:

Roles

We found that 85% of the participants in the Dev group and 96% of those in the PM group demonstrate a rather proactive stance towards self-fixing TD, i.e., they repay their own TD at least when necessary. The result of a Fisher's exact test shows that

the difference observed between the two groups is significant (p -value = 0.01). Although a similar disparity can be observed in Figure 5.9, a Fisher's exact test shows it is not substantial. Altogether, the findings suggest that **people in both types of roles engage in self-fixing TD, although those in project management and support seem to do it more often.**

Software development experience

After applying pairwise Fisher's exact tests with Bonferroni-Holm correction, we found no statistically significant difference between the three groups, i.e., **the level of development experience seems to have no effect on how participants repay their own TD.**

Contributions to the projects

For the groups with various levels of contribution, we found that 98% of the participants with high contribution level (ContribHigh) have a positive attitude towards self-fixing TD, and that 90% of them mentioned that they have *definitely* or *probably* self-fixed TD. These percentages were higher than the proportions for practitioners with medium (ContribMed) and low (ContribLow) contribution levels (87%/55% and 80%/43% respectively). Similarly, we conducted Fisher's exact tests for the three possible pairwise comparisons (ContribHigh versus ContribMed, ContribHigh versus ContribLow, and ContribMed versus ContribLow). After applying the Bonferroni-Holm correction, we found that participants with high contribution level (ContribHigh) differ significantly from the other two groups (ContribMed and ContribLow) in all of the comparisons (p -values < 0.05).

These findings suggest that **practitioners with high contribution level display a significantly more positive attitude towards self-fixing TD.** In addition, more than two-fifths of the participants with low contribution level (42%) and about one-third of the participants with medium contribution level (27%) mentioned that they have not self-fixed TD in the projects. As this difference is significant (p -value = 0.04), we identified further evidence that **the level of contribution reflects the self-fixing attitude, both positively and negatively.**

Project experience

For groups with different levels of experience in the projects, we found that 93% of participants with higher experience (ExpProHigh) consider fixing their own debt at least when necessary. The percentage for those with medium and lower experience (ExpProMed and ExpProLow) is 89% and 79%, respectively. The results of

the Fisher's exact tests with Bonferroni-Holm correction suggest a significant difference between participants with high and low experience (p -value = 0.02), and no significance between these two groups and ExpPromed (medium experience). This finding is further supported by the differences observed in Figure 5.9, which are all statistically significant. Altogether, these results suggest that **participants who have been involved in the projects for a longer period also seem to be more keen to repay their own TD.**

5.5 Discussion

5.5.1 Interpretation of Results

To the best of our knowledge, this is the first study to inquire practitioners on how they repay their own debt. Moreover, we argue that this work is complementary to previous studies that used source code analysis. Thus, we start our discussion with the results that can be compared more directly.

One important finding is that, although most Java and Python practitioners acknowledge that they have self-fixed TD, Java participants were slightly more certain. This may be due to the fact that, among the five debt types, 76%–84% of Java participants acknowledged that they have noticed similar debt, while the proportions are 53%–71% for Python participants. We also found that although all participants are likely to acknowledge that they have self-fixed Code Debt, there are some differences between Java and Python practitioners. For example, the former are similarly concerned about Test Debt, while Python participants have self-fixed Test Debt the least. The results are consistent with the findings obtained from source code level, i.e., Test Debt is commonly self-fixed in Java (Liu et al., 2021), while Test Debt appears to show the lowest self-fixing rate among the five debt types in Python (Tan et al., 2020). We also found that only 4% of Java participants mentioned that they have not noticed Test Debt, while the percentage of Python participants is over 26%. One possible explanation may be related to the population of the study: the percentage of Java participants with higher experience and contribution level is about twice that of Python participants; since repaying test smells is time-consuming (Bavota et al., 2015), it may be more commonly performed by experienced developers.

Regarding the reasons to introduce and self-fix TD, one can notice that non-technical reasons seem to be the most cited ones. Almost three-quarters of the reasons for introducing TD are related to *planning and management*, *lack of knowledge*, *people* and *organizational*, while most reasons to self-fix TD are related to *planning and management*, *people* and *knowledge*. The observations related to introducing TD are consistent with the literature (Rios et al., 2020). There is no literature to compare

the self-fixing results, but we do observe that these reasons are quite similar to those for introducing TD.

Despite the dominance of non-technical reasons, we found that technical reasons were frequently mentioned together with non-technical ones. For example, the most recurrent relations involve reasons to introduce TD related to *methodology* and *planning and management*. The latter (non-technical) reason is also the most often associated with reasons to self-fix TD related to *development*. This intertwining between technical and non-technical reasons reflects the multi-faceted decision-making process related to repaying one's own debt. Moreover, we found that practitioners might introduce TD because of *debt inheritance*, which is consistent with a related finding that developers are frequently forced to introduce new TD due to already existing TD (Besker, Martini and Bosch, 2018) ("the poor are getting poorer").

Furthermore, the most cited reason to self-fix TD is the practitioners' *sense of responsibility*, which indicates the relevance of further investigating the impact of human factors on self-fixing TD. The other two most cited reasons for self-fixing TD are to *avoid higher cost* and *improve maintainability*, which indicate a consideration of the trade-off between cost and benefit in order to find the appropriate moment to self-fix TD. This result aligns with related findings that mentioned *focus on short-term goals* and *cost* as reasons when participants are considering whether or not to repay TD (Freire et al., 2020).

Finally, the results also suggest that the practitioners' level of involvement (i.e., experience in the project and contribution level) can be related to how they deal with their own TD. This observation may be at least partially explained by a sense of code or project ownership: those with higher levels of involvement were significantly more positive towards self-fixing. Furthermore, practitioners in a management role were also found to display a more positive attitude. To assess the impact of practitioners' responsibility on repaying their own TD, we divided the participants into two groups based on whether or not they mentioned *sense of responsibility* as a reason to self-fix TD. The results show that 95% and 86% of the participants that did and did not mention responsibility, respectively, have self-fixed TD at least when necessary, although this difference is not statistically significant. In addition, all of the participants that mentioned *sense of responsibility* acknowledged that they have *definitely* or *probably* self-fixed TD, compared to only half of the other participants. We assessed this disparity with a Fisher's exact test and confirmed it to be significant. Thus, the sense of responsibility is an important driver to self-fix TD and should be properly motivated and rewarded.

5.5.2 Implications to Researchers and Practitioners

The results suggest that Java and Python practitioners have different attitudes towards self-fixing different types of TD. Thus, researchers can develop tools to prioritize TD remediation by assigning different weights to different TD types and giving suggestions to developers. For example, Test Debt and Defect Debt can be prioritized along with Code Debt for Java projects. In addition, our findings advance the state of the art as they expand the currently-known list of reasons to introduce TD and provide a list of reasons to repay one's own TD. Broader knowledge of such reasons can lead to more informed decisions when designing future studies on TD. For example, the detailed reasons to introduce and self-fix TD can help researchers in devising and evaluating prevention strategies for the most cited reasons for introducing TD and more timely management strategies for the most cited reasons to self-fix TD. One concrete suggestion is to encourage admission of TD (especially incurred under pressure) and explicitly allocate time to assess the current risk of existing TD.

We learned that practitioners are often willing to address their debt, displaying a sense of responsibility and collaboration. Also, it seems that repayment decisions are not made light-heartedly (e.g., due to pressure or at every single chance) but by weighing various factors, such as time, expected quality and confidence of efficacy. Thus, team leaders should encourage their team to admit TD, discuss it and document the corresponding items in their backlog. We also hope that practitioners gain a broader view of possible reasons to introduce and self-fix TD and how common they are; this may help them to make more informed decisions when managing TD and improving software quality. For example, team leaders can strengthen team communication and promote collaboration, which can boost self-fixing as we observed that *"improving collaboration"* is a recurrent reason to self-fix TD. Finally, software engineers should be given incentives to fix their own TD as a means of increasing their technical knowledge, which was also observed as a reason to self-fix TD (*experience and knowledge increased*).

5.6 Threats to Validity

Construct validity regards the connection between the research questions and the study objects. Although the survey participants are at the center of the study design, the questionnaire responses are the medium through which we capture their perception. Thus, the study's findings depend on how the participants interpret the questions and how we interpret the responses. To mitigate potential bias in interpretation, we provided definitions of the necessary TD-related concepts. Also, the level

of clarity in answers to open-ended questions (e.g., reasons to self-fix TD) confirms to some extent that the participants understood the underlying concepts.

Furthermore, we used log-transformations in Scott-Knott Effect Size Difference (ESD) test to mitigate the skewness of data distribution; as pointed by Tantithamthavorn et al. (2016), different results might be obtained if other transformations were used (e.g., Blom (Mittas and Angelis, 2013) or Box-Cox (Azzeh et al., 2015) transformations). This limitation is to some extent addressed by repeating each test multiple times to assess if the distribution of bias and variance are approximately normally distributed (as per the Central Limit Theorem) (Moore, 2018).

External validity concerns threats to the generalizability of our findings. Given the population of our study, one cannot expect the results to apply to every development team. However, we believe that our findings can be extrapolated to a considerable portion of the Java and Python open-source communities, since our study included participants from popular open-source projects in various domains and with diverse team sizes. While Python and Java are the two most popular programming languages, we cannot generalize our results beyond these languages.

Reliability considers the bias from the researchers in data collection or data analysis. To mitigate bias from the interpretation of open-ended responses (i.e., reasons), we applied open coding in two steps to categorize the replies. Specifically, the first two authors manually classified different reasons of introducing and self-fixing TD. To measure the level of agreement between the classifications of independent researchers, we estimated the inter-rater agreement using Krippendorff's alpha (Krippendorff, 2011) to be 0.88³, which was calculated based on the classification of all reasons. In the conflicting cases, the first and second authors discussed with the third author until they achieved consensus.

5.7 Conclusions

This chapter reports an empirical study investigating the phenomenon of self-fixed technical debt, i.e., when developers repay TD items that they have introduced themselves. In particular, we surveyed 181 practitioners spread among 17 projects (written in Java or Python) to explore participants' perception of this phenomenon and under what circumstances they self-fix TD.

The results show that most practitioners display a positive attitude towards self-fixing TD, and seem mostly attentive to Code Debt. Moreover, we found that non-technical reasons (e.g., related to planning and management) are the most considered both when introducing and self-fixing TD, and are often mentioned together

³Krippendorff's inspection of the tradeoffs between statistical techniques establishes that it is customary to require $\alpha \geq 0.80$ (Krippendorff, 2018).

with technical reasons (e.g., related to methodology). In particular, many developers mention a sense of responsibility as a factor for self-fixing, and that decisions to repay are not made easily but by balancing costs and benefits, among other factors.

Having investigated TD remediation from source code analysis and developers' perspectives, we noticed that more research is needed. In particular, the complete timeline of TD remediation has not been thoroughly examined since studies often look at a single data source (e.g., source code or developers' conversations). Thus, the next chapter focuses on investigating how TD items are managed in both issue trackers and source code.

Chapter 6

Complete Debt Stories as Told by Source Code and Issue Tracker: An Empirical Study of Technical Debt Timelines

Abstract

Although Technical Debt (TD) has gained more attention in recent years, most studies exploring TD are based on a single source (e.g., source code, code comments or issue trackers). Investigating information combined from different sources may yield insight that is more than the sum of its parts. In particular, we argue that exploring how TD items are managed in both issue trackers and source code can shed some light on what happens between the commits that incur TD and those that pay it back. For that, we randomly selected 3,000 issues from the trackers of five projects, manually analyzed 200 issues that contained TD information, and identified and investigated 219 TD item timelines. The results indicate that most of the TD items marked as resolved in issue trackers are also paid back in source code, although many are not discussed further after being identified in the issue tracker. Design Debt items are more likely to be paid back in source code, while Test Debt items are the least likely. We also learned that although TD items may be resolved a few days after being identified, it often takes a long time to be identified (around one year). In general, time is reduced if the same developer is involved in consecutive moments (i.e., introduction, identification, discussion and remediation), but the number of people involved in discussing the TD item does not seem to affect how quickly it is resolved.

6.1 Introduction

In recent Technical Debt (TD) management research, a number of data sources have been investigated, including source code (Tan et al., 2021; Digkas et al., 2020) and textual descriptions written in natural language such as source code comments (Maldonado et al., 2017) and issue trackers (Li et al., 2020). Most of the studies exploring text-based descriptions have focused solely on a single source (e.g., source code comments or issues), while few studies have looked at TD repayment using a

combination of sources (Zampetti et al., 2018). In particular, to the best of our knowledge, there is still a lack of research exploring how TD items are managed in both issue trackers and source code. This has two advantages over studying TD through individual sources.

First, while previous studies have shown that the repayment of a large percentage of TD is documented in issue trackers (Li et al., 2020; Xavier et al., 2020), it is not always true that the TD is paid back. For example, if the remediation of a TD item is added to a patch, and the patch is rejected, the item is not actually fixed in the project. Thus, the issue has been seemingly resolved in the issue tracker, while the code still remains unchanged. This begs the question of whether it was not necessary to fix the TD item in the code or there was a decision to keep the unchanged code. To address the shortcoming, we propose to examine commit messages and the source code to determine if the developers have actually paid back the TD item in the project.

Second, by focusing only on the source code (e.g., source code comments) one may also get false positives (Bavota and Russo, 2016; Maldonado et al., 2017). For example, the deletion of comments does not imply the remediation of the debt. Also, comments can be removed accidentally when entire classes or methods are deleted (Zampetti et al., 2018). However, one can know more clearly whether developers decide to resolve TD or not by analyzing issue documentation in the issue tracker, because developers often discuss TD items and possible solutions to them. Since remediation can happen as a ripple effect of maintenance, exploring the developer's intent can help better understand to what extent and how developers prioritize TD remediation. In this way, we can compare developers' intent to repay back TD against their actual actions in source code.

We argue that combining information from different sources can yield insight that is more than the sum of its parts. Using both source code and issue trackers, we can shed some light on what happens between the commits that incur TD and those that pay it back. For example, we can find out *when* developers discovered the TD item and *how* they decide to resolve it. Figure 6.1 depicts the typical timeline of a TD item evolution combining both source code and issue trackers. The white boxes represent the moments in the timeline of each TD item and the gray dashed boxes represent where the moments take place, i.e., source code or issue trackers. The black numbered labels of the arrows represent the time interval between different moments in the TD item timeline. TD introduction and remediation are the moments when TD items have been introduced and paid back in source code, and TD identification refers to the moment when TD items have been first identified in issue trackers. However, after identifying TD items, developers would have several discussions to determine if and how to pay back them in source code. In Figure 6.1,

TD discussion refers to the moment when developers finally decide to pay back the TD items.

Figure 6.1: TD timeline in issue trackers and source code.

Analyzing how developers introduce, identify, discuss and pay back TD at different moments within both source code and issue trackers can help better understand the overview of how developers manage technical debt. In addition, since software development is a highly collaborative activity, developers can use issue trackers and commit messages to effectively convey useful information such as the rationale of decisions. Thus, looking at how developers involved in issue trackers and commits influence the management of TD items can also provide a more complete understanding of why TD is being paid back.

In that context, this chapter aims at investigating the timeline of TD items incurred in source code, reported and discussed in issue trackers and then paid back in source code. In particular, it examines the likelihood of issues resolved in issues trackers also being resolved in source code, the length of the intervals between the various moments in the TD item timeline, and how the number and involvement of developers are related to the timeline. To this end, we extracted 3,000 random issues and analyzed TD information in almost 15K pieces of relevant issue documentation (i.e., issue summaries, descriptions and comments) from five open-source projects. Then, extracted 219 TD timelines through the manual inspection of 200 random issues.

To analyze the timelines, we collected code commits together with commit messages from GitHub, and used a regular expression-based approach (Fischer et al., 2003) to match each analyzed issue that contained TD to the corresponding commit. Next, we used *git blame* to trace the first commit in which developers introduced the pieces of code incurring TD. Finally, we calculated the intervals between the various moments in TD items timeline and extracted the corresponding developers' information.

The results show that most of the TD items identified and mentioned as resolved in issue trackers are also paid back in source code, and that about one-fifth are discussed further after being identified in the issue tracker. TD items commonly stay dormant during most of the timeline, being resolved within a few days of its identification. Some types of debt (e.g., Design Debt) are more likely to be paid back than others (e.g., Test Debt). Finally, although the involvement of one same developer in introduction, identification and repayment tends to lead to quicker turnarounds (e.g., faster identification or repayment), the way developers are involved in the issue tracker discussions does not seem to impact the length of time intervals. For example, whether the developer who paid back the item also participated in the issue tracker discussion has no significant effect on how long it takes to resolve the item.

The remaining sections are organized as follows. The chapter starts by summarizing related work and contextualizing our study in Section 6.2. The study objectives, the research questions, and provides details regarding the data collection and analysis are reported in Section 6.3. Section 6.4 presents the results of our study and Section 6.5 discusses the results' interpretation and implications. The threats to the validity of our study are reported in Section 6.6, and our final thoughts are presented in Section 6.7.

6.2 Related Work

In this work, we investigate how developers introduce, identify, discuss and pay back TD based on both source code and issue tracker information. Thus, this section presents the state of the art regarding the exploration of these two resources and how developers manage TD based on them.

A large number of studies focused on investigating TD introduction and remediation from source code. For example, some recent findings have shown that a large percentage of TD is paid back during software evolution (Tan et al., 2021; Digkas et al., 2018; Lenarduzzi et al., 2019; Baldassarre et al., 2020). Some studies focused on investigate TD items that are introduced intentionally and explicitly documented in code comments, i.e., self-admitted technical debt (SATD) (Potdar and Shihab, 2014; Bavota and Russo, 2016; Fucci et al., 2020). Another line of research considers the information in commit messages that are related to TD. Zampetti et al. (2018) analyzed commit messages in five Java projects and found that 8% of the self-admitted technical debt removal is acknowledged in commit messages. They randomly selected 300 pairs of commit message and SATD comment, and manually labeled all of them ranging from "not similar at all" to "very similar". Lu (2019) analyzed 33

Apache Java projects and found that the certain content of commit messages, e.g., empty messages are potentially associated with technical debt.

More recently, a few studies focused on investigating the identification and repayment of SATD in issue trackers. For example, Li et al. (2020) investigated the identification and repayment of SATD in issues from two large open source projects and found that on average 71.7% of identified TD is paid back, and most of it by TD identifiers and creators. In addition, Bellomo et al. (2016) manually examined 1,264 issues in four issue trackers from two government projects and two open source industry projects. Finally, they classified 109 issues as technical debt issues and derived generic characteristics for these issues.

In the work reported in previous chapters, we have also exclusively focused single sources of information and have investigated particular phenomena of TD (i.e., co-occurrence and self-fixed technical debt). Main findings include that developers tend to address their own TD and that self-fixed TD items are likely to have a shorter survival time. In comparison, related findings indicate that the majority of SATD is removed by the same developer who introduced it (Maldonado et al., 2017; Bavota and Russo, 2016; Liu et al., 2021).

6.3 Study Design

This empirical study was designed based on the guidelines of Runeson et al. Runeson et al. (2012) and is reported according to the Linear Analytic Structure Runeson et al. (2012).

6.3.1 Objectives and Research Questions

The goal of our study, described according to the Goal-Question-Metrics (GQM) approach (Van Solingen et al., 2002), is to “*analyze software systems for the purpose of investigating the timeline of technical debt items resolved in issue trackers with respect to the likelihood of them being also resolved in source code, the length between the various moments in the timeline, and how the number and involvement of developers are related to the timeline from the point of view of software developers in the context of open source software*”. This objective is further refined in terms of the following research questions.

RQ1: To what extent technical debt items identified and discussed in issue trackers are paid back in source code?

Specifically we focus on investigating to what extent resolved TD items can be mapped to the commits that address them. To this end, we first match each resolved issue with the messages in all the commits to find the corresponding com-

mit(s). Then, we manually check if TD-related problems in the issue documentation (i.e., summary, description, and comments) are resolved in the associated commits. Investigating this research question provides insight into the likelihood of the TD-related issues discussed in issue trackers actually being resolved in the code. This gives a more accurate view of TD remediation and can help project managers to further monitor TD management activities and review issue documentation practices. They can also help researchers design and refine tools to support TD tracking in issue trackers.

RQ2: How long are the time intervals between introducing, identifying, discussing and paying back TD items?

In this research question, we consider three time intervals in TD management (①②③, as shown in Figure 6.1). Technical debt can be intentionally or unintentionally introduced into source code by developers, and then reported by those who introduced it or others in issue trackers. In this context, interval ① refers to how long it takes to report the technical debt after introducing it in source code. Next, developers would have several discussions and then determine if and how to pay back TD in source code. Interval ② refers to how long developers take to decide how to resolve the technical debt after being reported in the issue tracker. Finally, with a plan defined, developers will take action and refactor the code accordingly. In the best case scenario, developers will apply changes in a short time. However, developers may also resolve it later in some cases, e.g., when the TD item is not urgent or needs to be resolved by a specific developer. Thus, when a TD item is actually solved in source code, it is important to also investigate interval ③, i.e., how long developers take to pay back technical debt in source code after a solution is planned.

In summary, this research question focuses on investigating the intervals between the various moments in a TD item timeline. Such an investigation fills the gap of studying TD remediation solely on the source code, and provides a more complete picture of how and when developers manage technical debt by analyzing their identification and discussion in the issue trackers, together with their introduction and remediation in the source code. Moreover, project managers can pay more attention to intervals that are growing relatively long.

RQ3: How can the number and involvement of developers associate with the management of technical debt items documented in issue trackers?

In this research question, we seek to identify the number of developers associated with the length of the three timespans in TD items timeline (i.e., ①②③) and the overall survival time. In particular, we distinguish between two types of developers: those involved with a single source of evidence (i.e., source code or issue tracker), and those involved in both source code and issue tracker. To answer this

question, we consider the number of each type of developer in the four moments. For example, for the discussion moment we would look into the number of developers that have only been involved in the issue tracker discussion, and the number of developers that have been involved in the discussion and have also authored commits that introduced the debt item. Such findings can help developers more reasonably participate in TD management and improve the efficiency of TD remediation. For example, if more developers participating in a tracked issue can speed up TD remediation, then project managers can identify potentially relevant developers and encourage them to engage in the discussions.

6.3.2 Project Data Extraction

To perform this study, we focused on the Apache Software Foundation, since Apache projects are developed and maintained by technical experts following Apache-wide meritocratic rules, and thus, they have a reputation for high quality and long-term stability¹. In addition, Apache projects cover a wide range of languages, sizes and number of contributors, since Apache Software Foundation is the world's largest open source foundation and contains more than 300 top-level projects². Furthermore, Apache projects are commonly selected in other technical debt studies (see e.g. Potdar and Shihab (2014); Bavota and Russo (2016); Li et al. (2020)). The projects selection process was according to five inclusion criteria:

1. The project must be publicly available (including the repository and the issue tracker), and it must also use Git as the version control system and JIRA as the issue tracker.
2. The project must be under development for at least five years and have more than 1000 commits. A long history enables developers to observe the severity of TD and fix it, and this number of revisions provides a sufficient set of repeated measures for statistical analysis Digkas et al. (2020).
3. The project must have 200,000 source lines of code (SLOC), and 5,000 issues in the issue tracker. Larger projects typically cover more TD types, and enough issues indicate that developers are using the issue tracker frequently and actively.
4. The project must involve at least 100 developers, ensuring a diversity of human factors (e.g., seniority) among developers.

---

¹<http://www.apache.org/foundation/how-it-works.html>, visited February 2021

²<https://www.apache.org/index.html#projects-list>, visited February 2021

Table 6.1: Details of the Five Selected Projects

Project	Project Details			
	Languages	Commits	SLOC	Contributors
Camel	Java	50,380	1,525k	722
Hadoop	Java	24,612	3,409k	22,954
Hbase	Java	18,340	912k	2,992
Impala	C++, Java, and Python	9,661	640k	149
Thrift	C++, Java, and C	6,197	294k	353

5. The project must require developers to include JIRA ticket numbers in the commit messages when providing code patches (e.g., Apache Camel³). This is necessary to support linking the issues (in the issue tracker) to the commits (in the source code repository).

To enhance the representativeness of the sample (and mitigate threats to external validity), we randomly selected five projects based on the aforementioned criteria. Table 6.1 presents the details of each project, including the most popular languages that contributors used, the number of commits, SLOC (source lines of code) and the number of contributors. We analyzed the latest released versions on February, 2021.

6.3.3 Variables and Data Collection

This section provides the sets of variables necessary to answer the research questions and the major steps of the data collection process. In particular, each unit of analysis consists of a TD item timeline, which encompasses: (a) the commit(s) that *incurred* the debt; (b) the issue documenting the *identification* and *discussion* of the debt item; and (c) the commit(s) that *pay back* the debt.

The issue (b) includes the *issue keys*, *issue sections* (i.e., summary, description and comments), *resolution status*, *developers* who reported, discussed and resolved the issues, *creation timestamp*, *resolution timestamp* and the *types of technical debt*. The commits corresponding to the analyzed issue (a and c) include *repository name*, *commit hashcode*, *commit date*, *author's name*, *author's email* and *commit message*. Data collection is comprised of four main parts: (i) extracting and filtering issues, (ii) manually identifying and classifying technical debt, (iii) matching the analyzed issues to the commit(s) that pay back the TD item, and (iv) determining the commits that incurred the analyzed TD item.

Step i: Extracting and Filtering Issues

³<https://camel.apache.org/manual/latest/contributing.html>, visited February 2021

Since we are aiming at investigating the complete timeline of TD items, we start by collecting the resolved issues, i.e., those with status marked as ‘resolved’ or ‘closed’. Since manual analysis of the issues is extremely time-consuming, we can only analyze a subset. Thus, we randomly select 600 issues from each project, totalling 3,000 issues from the five projects.

Step ii: Manually Identifying and Classifying TD

Similarly to Li et al. (2020), we extract the sections of tracked issues (i.e., summaries, description, and comments) and analyze them at the sentence level. In particular, we search sentences that indicate technical debt, and tag the debt status in each one (i.e., paid back or not). To classify TD, we use the classification framework from Li et al. (2020) and remove TD types that are not closely related to the source code (e.g., Requirement Debt). This framework is also used to classify the types of TD contained in issues (see Table 6.2), which is based on a well-known TD classification framework provided by Alves et al. (2014).

Table 6.3 reports the issue and technical debt details for each project, including the total number of issues, the number of sections extracted from the selected 600 issues, and the number of issues and sections that contained TD.

Step iii: Matching Issues to Remediation Commits

According to the project selection criteria (see Section 6.3.2), all five projects require developers to add the JIRA issue key when providing code patches. Thus, we clone the *git* repository of each project, and use *git log* to extract the code commits together with the corresponding commit messages. The commit messages typically contain an issue ID, indicating the issue that each commit addresses. With this information, we use the regular expression-based approach provided by (Fischer et al., 2003) to match the issue key in the commit message.

Table 6.4 shows the number of issues for which the ID can be mapped in the commit messages per TD type. It is worth noting that there are only five Defect Debt items in total, so we will not consider that debt type in the following analysis. Moreover, the manual review the code that resolved the TD item and further trace back to the code that introduced the same item is fairly time consuming. Thus, we limited the analysis to delving into 200 issues. According to Table 6.4, the number of issues are unequally distributed across the projects. Thus, we randomly selected issues by using *stratified random sampling* (Shull et al., 2007), which is used to estimate population parameters efficiently when subpopulations have substantial variability (Cochran, 2007). For each project, we randomly selected a number of issues based on the total number of issues it contains.

Developers can identify TD in issue trackers in three cases: before creating an issue, during code review and after a patch is committed (Li et al., 2020). Since this study considers a timeline starting with the TD introduction in the source code,

Table 6.2: Different TD Types that are Detected in Issue Trackers

Types	Indicator	Definition
Architecture Debt	Violation of modularity (VioMod)	Because shortcuts were taken, multiple modules became inter-dependent, while they should be independent.
	Using obsolete technology (ObsTech)	Architecturally-significant technology has become obsolete.
Code Debt	Complex code (CpxCd)	Code has accidental complexity and requires extra refactoring action to reduce this complexity.
	Dead code (DedCd)	Code is no longer used and needs to be removed.
	Duplicated code (DupCd)	Code that occurs more than once instead of as a single reusable function.
	Low-quality code (LQualCd)	Code quality is low, for example because it is unreadable, inconsistent, or violating coding conventions.
	Multi-thread incorrectness (MTCor)	Thread-safe code is not correct and may potentially result in synchronisation problems or efficiency problems.
	Slow algorithm (SIAlg)	A non-optimal algorithm is utilised that runs slowly.
Defect Debt	Uncorrected known defects (Def)	Defects are found by developers but ignored or deferred to be fixed.
Design Debt	Non-optimal defects (Des)	Non-optimal design decisions are adopted.
Test Debt	Expensive tests (ExpTst)	Tests are expensive, resulting in slowing down testing activities. Extra refactoring actions are needed to simplify tests.
	Flaky tests (FlaskTst)	Tests fail or pass intermittently for the same configuration.
	Lack of tests (LacTst)	A function is added, but no tests are added to cover the new function.
	Low coverage (LCvg)	Only part of the source code is executed during testing.

Table 6.3: Issue and Technical Debt Details of the Five Selected Projects

Project	Issue Details		Technical Debt Details	
	Issues	Filtering Sections	TD Issues	TD Sections
Camel	14,411	1,661	209	268 (16.1%)
Hadoop	16,808	2,785	371	567 (20.4%)
Hbase	24,342	4,946	336	743 (15.0%)
Impala	9,733	1,973	272	381 (19.7%)
Thrift	5,196	2,763	313	435 (22.0%)

Table 6.4: Number of Issues per TD Type

Types	Number of Issues					
	Camel	Hadoop	Hbase	Impala	Thrift	Sum
Architecture Debt	14	11	6	4	6	41
Code Debt	64	120	107	56	129	476
Design Debt	80	34	82	74	75	345
Test Debt	30	36	46	25	29	166
Total Issues	155	164	184	152	202	
Selected Issues	36	38	43	36	47	

we only consider the first case (i.e., TD is the reason for creating the issue), and manually filter out the others. Based on the above criteria, we finally selected 36–47 issues per project (as shown in the bottom of Table 6.4) .

We note that there may exist multiple commits for fixing one issue. One possible reason is that an issue may be too complex and developers have to divide the fix of this issue into several commits. Another possible reason is that developers might find that an issue was actually not entirely fixed, reopened and fixed it in a later commit (Xia et al., 2015). In these cases, we consider all related commits.

Step iv: Determining TD Introduction Commits

To determine the commits that incurred TD, we first identify the pieces of code that contain the TD being resolved. Then, we use *git blame* to trace the modification history of these pieces of code, and consider the commit that created the code to be the TD introduction commit.

6.3.4 Data Analysis

For **RQ1**, to investigate to what extent technical debt items resolved in issue trackers are paid back in source code, we first identify the selected issues that contain TD items. Then, we match the issues to the corresponding commits by using the method

provided in Section 6.3.3. Finally, we manually check if TD-related problems in the issue documentation (i.e., summary, description, and comments) are resolved in the corresponding commits.

To answer **RQ2**, we need to determine the four moments in the TD item timeline before we calculate the three time intervals between them (①②③, as shown in Figure 6.1). Regarding the moment of TD introduction, we first use *git blame* to trace the first commit in which developers introduced the pieces of code incurring TD. The commit date of that commit is the TD introduction moment. Regarding the TD identification moment in issue trackers, we use the date in the issue documentation where TD-related problems are first mentioned. For interval ①, we use the time difference between the moment of TD identification and TD introduction.

After TD-related problems are identified and flagged in issue trackers, developers may have several discussions about TD before they decide to resolve it. To represent the moment of TD discussion, we use the date in the issue documentation when developers mentioned their final decision about the remediation of the TD items. For interval ②, we use the time difference between the moment of TD discussion and TD identification.

Since we focus on TD items that have been resolved in both issue tracker and source code, after developers decide how to resolve TD in issue trackers, the next step for them is to resolve the corresponding TD-related problems in the source code. To define the moment of TD remediation, we use the date of the last commit that has been determined to address the tracked issue. Thus, for interval ③, we use the time difference between the moment of TD remediation and TD discussion.

In **RQ3**, to identify how the number of developers associate with the length of the three timespans in TD items timeline (i.e., ①②③), we first consider which developers were involved in the four moments. For that, we create unique, anonymous ID's for each developer involved in relevant commits (i.e., incurring or paying back TD) based on their name and email.

Next, we focus on interval ① and investigate whether the length of this timespan is related to whether the developer identifying the debt item (in the issue tracker) also authored a commit that introduced (part of) the debt item. Then, we focus on interval ② and examine the total number of developers involved in the issue tracker discussions, the number of developers involved in the discussions who have also authored commit(s) that introduced (part of) the debt, and the number of developers who have been involved in the discussion only (i.e., did not author any commit that incurred the debt item). For interval ③, we investigate whether the length of this timespan is related to whether the developer(s) paying back the debt in source code were also involved in the issue tracker discussions. Finally, we build a generalized linear mixed model (GLMM) (McCulloch and Neuhaus, 2005) to investigate the

relationship between the overall survival time and the mentioned developer-related factors.

6.4 Results

6.4.1 To what extent technical debt items identified and discussed in issue trackers are paid back in source code? (RQ1)

There are 396 TD sections (i.e., summary, description and comments) involved in the selected 200 issues, of which 15 sections are the TD introduced in patches during the discussion and then resolved in the final commits. Thus, we filtered out these TD items since they were not introduced in the source code. Overall, we have 381 TD sections from the selected 200 issues, which can be mapped into 249 TD items. The majority of these TD items corresponded to a single TD type, and only two corresponded to multiple types, which we discuss separately in the following.

One special TD item is found in the issue HBASE-1655 *Usability improvements to HTablePool*, this summary belongs to Design Debt. However, the description and the comments of this issue related to both Design Debt and Code Debt, the latter includes the indicators Low-quality code (LQualCd) and Dead code (DedCd). From the description:

- *Remove constructors that were not used.* - [Code-DedCd]
- *Change internal map from TreeMap to HashMap because I couldn't see any reason it needed to be sorted.* - [Design-Des]

And from the comments of issue HBASE-1655:

- *...The default getTable also requires instantiating a new HBaseConfiguration() each time internally, even if we are reusing an existing HTable... Whether there is a significant overhead or not to that, we should avoid it when unnecessary.* - [Code-LQualCd]
- *...only issues there are the tab issues, reordering, and whether to expose the pools or not.* - [Code-LQualCd]
- *Sorry about the tab/spaces issue. I didn't clean it up carefully enough.* - [Code-LQualCd]
- *Even though TreeMap uses the comparator rather than the equals method to compare keys, using a byte[] as the key seems to break the contract of a java.util.Map...* - [Design-Des]

- *...Seems better to just use String and HashMap which works well and satisfies the Map contract.* - [Design-Des]
- *While we lose the convenience of calling static methods...* - [Design-Des]

Another special TD item is found in the issue HBASE-18085, in which the description and the comments related to three indicators of Code Debt, i.e., Multi-thread incorrectness (MTCor), Dead code (DedCd) and Duplicated code (DupCd):

- *We observed threads blocking on the purge method while using offheap bucket cache...* - [Code-MTCor]
- *...using AtomicBoolean or volatile boolean we're actually re-implementing the tryLock logic* - [Code-DupCd]
- *...there is no logic other than just try lock and release. If that is been removed as dead code by compiler?* - [Code-DedCd]

Table 6.5 presents the results of the remaining 247 TD items (i.e., each of them only belonging to one TD indicator), including their types and indicators, and the numbers and percentages that are paid back in source code. The result shows that most of the TD items (i.e., 88.7%) identified and mentioned as resolved in issue trackers are also paid back in source code.

Regarding different types of TD, we found that Design Debt and Code Debt are more likely to be paid back in source code. To investigate whether one type of TD tends to be resolved more or less in source code than other types, we conducted pairwise Fisher's exact tests (Fisher, 1922) with Bonferroni-Holm corrections (Aickin and Gensler, 1996) on the percentages of TD items that are paid back in source code from the different types. The results show that the percentage of Design Debt items that are also paid back in source code differs significantly from Test Debt and Architecture Debt (p -values < 0.05). In addition, Test Debt items differ significantly from Code Debt. This finding suggests that the resolved TD items in issue trackers related to Design Debt have a significantly higher chance to be paid back in source code, while Test Debt items are less likely.

6.4.2 How long are the time intervals between introducing, identifying, discussing and paying back TD items? (RQ2)

During the data analysis, we found that developers often identified a TD item from the source code and then reported it in the issue tracker without any further discussion. Thus, it is hard to ensure when developers decide to resolve that TD item, i.e.,

Table 6.5: Numbers and Percentages of TD Items Resolved in Issue Trackers (IT) are Paid Back in Source Code (SC)

Types	Indicator	IT	SC	%	IT	SC	%
Code Debt	Low-quality code (LQualCd)	85	79	92.9	125	112	89.6
	Dead code (DedCd)	17	16	94.1			
	Slow algorithm (SlAlg)	14	10	71.4			
	Multi-thread incorrectness (MTCor)	7	6	85.7			
	Complex code (CpxCd)	2	1	50.0			
Design Debt	Non-optimal defects (Des)	90	85	94.4	90	85	94.4
Test Debt	Lack of tests (LacTst)	16	11	68.8	27	18	66.7
	Flaky tests (FlakTst)	6	2	33.3			
	Low coverage (LCvg)	4	4	100.0			
	Expensive tests (ExpTst)	1	1	100.0			
Architecture Debt	Using obsolete technology (ObsTech)	3	3	100.0	5	4	80.0
	Violation of modularity (VioMod)	2	1	50.0			
Total					247	219	88.7

the moment of TD discussion is unknown. To address this problem, after TD items were identified, we divided them into two groups according to whether the moments that the developers decided to resolve them could be determined. Figure 6.2 presents the updated timeline used to answer this research question, in which we renamed interval ③ (i.e., the time difference between the moment of TD remediation and TD discussion) to interval ③_a for the items with the moment of TD discussion. Interval ③_b refers to the time interval between the moment of TD introduction and TD remediation for all the TD items.

Figure 6.2: TD timeline

Among all 219 TD items that have been resolved in source code (as shown in RQ1), we found that for 41 of them the moment of TD discussion can be confirmed.

Figure 6.3: Distribution of Time Intervals between Introducing, Identifying, Discussing and Paying Back All TD Items

Figure 6.3 shows the box plots of the distribution of the number of days for intervals 1, 2, 3a and 3b. One can notice considerable differences in the descriptive statistics of the four intervals. In particular, the median values are 356, 1, 1 and 3 days for the four intervals, respectively.

To further evaluate the significance of such difference, we calculated the Kruskal-Wallis test (Breslow, 1970) (p -value < 0.01) on the number of days for the four intervals, since we cannot assume the intervals are normally distributed. The results reveal that the difference between the four intervals is significant (i.e., p -value < 0.05). This finding indicates that although TD items take a long time to be identified (around one year), they tend to be resolved in source code within a few days after being identified.

Then, we performed a separate analysis related to four considered TD types, i.e., Architecture Debt, Code Debt, Design Debt and Test Debt. Figure 6.4 shows the box plots of the distribution of the number of days needed to manage different types of TD for the four intervals. For each interval, we conducted the Kruskal-Wallis test to evaluate the significance of the difference between TD types. The results reveal no significant difference between different types of TD within the same time interval (i.e., p -value > 0.05). The finding suggests that, while the number of days varies widely between time intervals, there is no significant difference in how developers prioritize different types of debt within the same time interval.

Figure 6.4: Distribution of Time Intervals between Introducing, Identifying, Discussing and Paying Back Different Types of TD Items

6.4.3 How can the number and involvement of developers associate with the management of technical debt items documented in issue trackers? (RQ3)

Regarding interval ①, we investigated whether the length of this timespan is related to whether the developer identifying the debt item (in the issue tracker) also authored a commit that introduced (part of) that item. To answer this question, we used the Kaplan-Meier (K-M) method (Kaplan and Meier, 1958), which is a non-parametric statistic, to analyze time interval ① of all the 219 TD items. Figure 6.5 shows the Kaplan-Meier plot to visualize timespan curves of all the TD items, in which depicts the cumulative percentage of TD items (y-axis) that have been identified within a specified number of days (x-axis).

Figure 6.5: Results of Kaplan-Meier method for the time interval 1 of all the 219 TD items

In Figure 6.5, if one item was identified and marked in the issue tracker by the same developer who introduced it in the source code, we called it a self-identified item; otherwise, it is a non-self-identified item. In addition, the number of TD items that have not yet been identified is shown at the bottom of the figure. For example, there are 193 non-self-identified and 26 self-identified items at the beginning of the projects, while 84 and 4 items, respectively, have not been identified for more than 500 days.

From Figure 6.5, we found that self-identified items were identified in a shorter period of time than non-self-identified items. To evaluate whether such a difference is significant, we conducted a log-rank test (Bland and Altman, 2004) on the length of the interval ❶ of self-identified and non-self-identified items. The result shows that the difference is significant (p -value < 0.01), which indicates that TD items seem to be identified faster in the issue tracker by the developers who introduced them in the source code.

Next, we focused on interval ❷ and examined the total number of developers involved in the issue tracker discussions (IT_Discuss), the number of developers involved in the discussions who have also authored commit(s) that introduced (part of) the debt (IT_Discuss_Intro), and the number of developers who have been involved in the discussion only, i.e., did not author any commit that incurred the debt item (IT_Discuss_Only). To examine these factors, we built a generalized linear mixed model (McCulloch and Neuhaus, 2005) (GLMM) to estimate whether the

Table 6.6: Results of the generalized linear mixed model

Diagnostics					
	AIC	BIC	logLik	deviance	df resid.
	41.5	51.8	-14.7	29.5	35
Scaled residuals					
	Min	1Q	Median	3Q	Max
	-0.5	-0.5	-0.4	1.6	4323.7
Random effects					
Groups name	Std.Dev.				
TD item ID (Intercept)	0				
Project ID (Intercept)	0				
Quality metrics					
Metric	OR	Estimate	Std.Error	z-value	p-value
(Intercept)	1.645e-01	-1.805	1.610	-1.121	0.262
IT_Discuss	6.925e-01	-0.368	1.780	-0.206	0.836
IT_Discuss_Intro	2.007e-08	-17.724	4882.327	-0.004	0.997
IT_Discuss_Only	2.394e+00	0.873	2.999	0.291	0.771

length of interval ② correlates with how developers were involved in the issue tracker and the source code.

In the model, the random effect is represented by the TD item ID, and helps to account for differences between different items in terms of three factors. Moreover, the value of the independent variables can depend on projects' characteristics, e.g., the longer the project survived, the more developers might be involved in it. To properly interpret the importance of each variable in the model, we use min-max normalization to normalize variable values, within each project, in the interval $[0,1]$ ⁴. After that, we built the GLMM using the *glmer* function of the *lme4* R package (Bates et al., 2007).

Table 6.6 reports the results of the logistic regression mixed-effect model, which show that the three aforementioned factors might have no statistically significant effect on the length of interval ②. In other words, how developers are involved in the issue tracker discussions seems to not influence the timespan in which the TD item is identified.

For interval ③, followed by RQ2, we also considered interval ③a (i.e., the time difference between the moment of TD remediation and TD discussion for the items

⁴This is done by subtracting the minimum and dividing by the difference between the maximum and minimum, which was also used in Iammarino et al. (2021).

Figure 6.6: Results of Kaplan-Meier method for the time interval 3a of all the 41 TD items

with the moment of TD discussion) and interval 3b (i.e., the time interval between the moment of TD introduction and TD remediation for all the TD items) in this research question. Regarding interval 3a, we investigated whether the length of the timespan is related to whether the developer(s) paying back the debt in source code were also involved in the issue tracker discussion. Regarding 3b, we investigated whether the length of the timespan is related to whether the developer(s) paying back the debt in source code also identified it in the issue tracker.

Figure 6.6 shows the Kaplan-Meier plot to visualize the timespan curves of the 41 TD items for which the moment of TD discussion can be confirmed. The plot depicts the cumulative percentage of TD items that have been paid back after the decision of being resolved (y-axis) within a specified number of days (x-axis). In Figure 6.6, if one item was paid back in the source code by the same developer who was involved in the issue tracker discussion, we marked it as an involved item; otherwise, it is a not-involved item.

As shown in Figure 6.6, the survival curves of involved and not-involved items almost overlap at the beginning, especially three quarters of involved items and not-involved items tend to be paid back quickly. However, all the involved items tend to be solved within one month after developers made the decision to pay them back, while not-involved items might take a longer time to be paid back. To evaluate the significance of the observed differences, we conducted a log-rank test on the

Figure 6.7: Results of Kaplan-Meier method for the time interval 3b of all the 219 TD items

length of the interval 3a of involved and not-involved items. The result shows no significant difference between the involved and not-involved TD items (p -value = 0.12). This result indicates that whether the developer who paid back the item also participated in the issue tracker discussion has no significant effect on how long it takes to resolve the item.

Figure 6.7 shows the Kaplan-Meier plot to visualize timespan curves of 3b for all the 219 TD items, depicting the cumulative percentage of TD items that have been paid back after being identified (y-axis) within a specified number of days (x-axis). In Figure 6.7, if one item was paid back in the source code by the same developer who identified it, we marked it as a self-resolved item; otherwise, it is a non-self-resolved item.

As shown in Figure 6.7, the majority of the self-resolved items and the non-self-resolved items have been fixed quickly, and the survival curves of those two kinds of items almost overlap at the beginning. However, the result from a log-rank test shows that the difference between the length of the interval 3b of self-resolved and non-self-resolved items is statistically significant (p -value < 0.01). Thus, TD items seem to be resolved faster in the source code by the developers who identified them and marked them in the issue tracker.

6.5 Discussion

In this section, we first elaborate on the interpretation of our results, and then we discuss the implications for researchers and practitioners.

6.5.1 Interpretation of Results

The Likelihood of TD to be Paid Back in Source Code

If technical debt items are identified and mentioned as resolved in issue trackers, they are most likely resolved in the source code as well, as we observed that 88.7% of them are paid back. Moreover, the likelihood of the resolved TD items (in issue trackers) to be paid back in source code varies widely among different types. For example, we found that Design Debt items have a significantly higher chance to be paid back in source code, while Test Debt items are less likely. One possible reason might be that developers realize the impact of Design Debt on software maintenance and try to address them to avoid paying high technical debt interest. This finding is consistent with the results reported in Chapter 2, i.e., Design Debt that is common in both Python and Java projects has a high fixing rate.

Survival Time

It is worth noting that although TD items take a long time (around one year) to be identified in the issue tracker, they are likely to be resolved in source code within a few days after being identified. In addition, only about one-fifth (i.e., 41/219) of TD items are further discussed and mentioned to be solved by developers after they have been identified in the issue tracker. The possible reason could be that developers usually identify the TD items that are considered more dangerous and prefer to solve them as soon as possible. However, regarding different types of TD, developers seem to treat them similarly, as we found no significant difference in how developers prioritize the different types of debt within the same timeline interval.

Human Factors

We found that TD items seem to be more quickly identified in the issue tracker by the same developers who introduced them in the source code. Such quicker turnaround pattern persists in later moments, e.g., when the same developer is involved in both identification and repayment. These findings are consistent with the results of the study reported in Chapter 4, i.e., developers are often willing to address their own TD, and the self-fixed TD tends to survive for a shorter time. One possible reason

might be that developers are more familiar with the TD items introduced or identified by themselves. However, how developers have been involved in the issue tracker discussions does not seem to affect the length of intervals. For example, whether the developer who paid back the item also participated in the issue tracker discussion has no significant effect on how long it takes to resolve the item.

6.5.2 Implications for Researchers and Practitioners

Investigating how developers introduce, identify, discuss and pay back TD at different moments within both source code and issue trackers can help researchers better understand the overview of how developers manage technical debt. For example, we found that TD items usually take a long time to be identified in issue trackers. However, once they are identified, they are likely to be resolved in source code within a few days. Thus, to improve the efficiency of debt repayment, researchers can develop tools to help developers identify and, more importantly, report TD items in source code.

We also hope that practitioners gain a broader view of technical debt repayment in source code. One suggestion is that when practitioners find a TD item in the source code, even if they have no plan to address it immediately, it is still advised to mention the item in the issue tracker. Our findings related to human factors can directly support practitioners to re-allocate suitable developers to projects. For example, for projects that are under time pressure, team leaders can suggest practitioners schedule focus on reviewing the components in which they have identified TD items since those practitioners are already familiar with the components and can address TD items in a shorter time.

6.6 Threats to Validity

In the following, we discuss the threats to construct and external validity of the reported study, as well as reliability threats.

Construct validity is related to the connection between the research questions and the objects of study. In this respect, TD items and timelines were manually validated to confirm the presence of the object of study. To avoid biases from investigating a limited sample size, we randomly selected the analyzed sample from all the collected issues.

External validity concerns threats to the generalizability of our findings. Given the population of our study, one cannot expect the results to apply to every development team. However, analyzed issues from five large open source projects with different programming languages and different size, and all of them use JIRA as the

issue tracker. Thus, we believe our findings can be relevant to other open source projects that use JIRA.

Reliability considers the bias from the researchers in data collection or data analysis. To address these threats, two researchers were involved in the data collection and analysis. Moreover, the first and second authors classified the TD items that contained in the issues independently into the different TD categories, using the description of the rules and the definition of the categories. To assess the disagreements numerically, we estimated the inter-rater agreement using Krippendorff's alpha (Krippendorff, 2011) ($\alpha = 0.82$)⁵.

6.7 Conclusions

This chapter reports on an empirical study that investigated the the timeline of technical debt items resolved in issue trackers. Especially, we investigate the likelihood of TD items being also resolved in source code, the length between the various moments in the timeline, and how the number and involvement of developers are related to the timeline. We randomly selected 600 issues from each project, totalling 3,000 issues from the five projects. Then, we manually analyzed 200 issues that contained technical debt items and used *git log* to extract the related code commits. Finally, we extracted 219 TD items from the analyzed 200 issues.

We found that most of the TD items identified and mentioned as resolved in issue trackers (i.e., 88.7%) are also paid back in source code, while about one-fifth (i.e., 41/219) are further discussed after being identified in the issue tracker. By looking at the timespan between the various moments in a TD timeline, results showed that items stay dormant for most of their existence and are often resolved quickly (within a few days) after identification in the issue tracker.

Regarding the different types of TD, Design Debt-related items have a significantly higher chance to be paid back in source code, while Test Debt items are less likely. However, there is no significant difference in how developers prioritize the different types of debt within the same timeline moment.

Finally, TD items seem to be reported faster when the developer who introduced them is the reporter, as well as resolved quicker when the developer who identified the items is the committer. However, the way developers are involved in the issue tracker discussions does not seem to affect the length of the timeline. For example, whether the developer who paid back the item also participated in the issue tracker discussion has no significant effect on how long it takes to resolve the item.

---

⁵Krippendorff's inspection of the tradeoffs between statistical techniques establishes that it is customary to require $\alpha \geq 0.80$ (Krippendorff, 2018).

Chapter 7

Conclusions and Future Work

This chapter presents the conclusions and contributions of the PhD dissertation, and also provides some directions for future work. In Section 7.1, we revisit the research questions and answer them according to the findings of the respective empirical studies. We also highlight the main contributions of this dissertation compared to the state of the art. In Section 7.2, we describe some perspectives for future work.

7.1 Research Questions and Contributions

As stated in Chapter 1, the main problem addressed in this dissertation is the following: *“If TD is not repaid, it can accumulate leading to severe quality problems, unexpected cost overruns and substantial financial loss. Due to the limited available resources, development teams need to make well-informed investment decisions on which TD repayment actions will yield the highest return. Thus, it is paramount to investigate how TD is paid back in practice and what factors influence this repayment, by considering both software repositories and the perspective of practitioners.”*. To tackle this problem, we asked five research questions in Chapter 1 and answered them in Chapters 2 to 6. In the following paragraphs, we repeat the questions and answer them based on the empirical evidence that we collected. Moreover, Table 7.1 summarizes the contribution of each chapter compared to the state-of-the-art.

RQ₁: *How is TD paid back over the course of time and what is the difference between Python and Java?*

To answer this research question, we performed a case study of the multi-year evolution of technical debt remediation in 44 Python projects of the Apache Software Foundation (ASF), focusing on the technical debt repayment effort and the issues that are fixed. The results showed that most of the repayment effort goes into improving testing, adding documentation, reducing complexity, and removing duplication. We also made a comparison between the results of this study and a study by Digkas et al. (2018) on Java projects of the ASF, presenting similarities and differences between the findings. Regarding the similarities, a minority

Table 7.1: Contributions of the PhD dissertation

Research Question	Chapter	Contributions compared to the state-of-the-art
RQ ₁ : How is TD paid back over the course of time and what is the difference between Python and Java?	2	This is the first study investigating the evolution of technical debt remediation in Python projects, regarding the fixing rate, remediation effort and remediation time. In addition, it is novel in comparing between the evidence in Python and Java projects and discussing commonalities and differences among the two languages.
RQ ₂ : What TD types are involved in co-occurring debt items and how long do they survive?	3	It is the first study to report on the relationship between co-occurring issues that belong to five different types of technical debt. In particular, it focuses on the issues and types of TD that co-occur frequently and on the interval between their introduction and removal.
RQ _{3,a} : What TD types are commonly self-fixed and how long does self-fixed TD survive?	4	This study has coined the term ‘self-fixed TD’ and reports the first evidence that exclusively focuses on self-fixed TD. It discusses the types of TD that are more likely to be self-fixed and their survival time, especially compared to non-self-fixed TD. The chapter also examines the relationship between self-fixing rate and factors related to the effort of developers and development teams.
RQ _{3,b} : Do practitioners intentionally self-fix TD and why?	5	It provides insights on the relevance of self-fixed TD to practitioners and the recurrence with which they address their own debt. Moreover, it elaborates on the circumstances that lead to self-fixing, also providing a list of reasons (both technical and non-technical) and highlighting how these associate with reasons to introduce TD.
RQ ₄ : What is the process between incurring a TD item and paying it back?	6	This study investigates how TD items are managed in both issue trackers and source code, which provides a broad view of what happens between the commits that incur TD and those that pay it back.

of rules account for the majority of issues fixed and the spent effort, and almost 90% of fixed issues and 80% of effort estimated to be invested in fixing, pertain to rules related to Design Debt, e.g., complexity. In addition, the majority of the TD items with the highest fixing rate (up to 60%) are language-specific issues, a minority of TD items can live for a long time. Regarding the differences: a) more than half of TD items in Python projects gets fixed is the short-term, i.e., they are paid back in less than two months, which is much faster than the previous evidence on Java projects; b) documentation-related debt seems more prominent in Python projects than in Java projects; and c) Test Debt tends to be paid back more frequently in Python. Despite these indications, we also highlighted that biases such as project-dependent variables (complexity, domain, etc.) may affect the comparison between the programming languages.

RQ₂: What TD types are involved in co-occurring debt items and how long do they survive?

To answer this research question, we carried out an empirical study that investigated the relationship between co-occurring technical debt types. We performed our study by analyzing the multi-annual evolution of TD in 20 Python projects (approx. 28,200 commits) from the ASF and identifying TD based on 42 rules defined in SonarQube. Moreover, we mined co-occurring TD based on a well-known machine learning algorithm, namely Apriori, in 3862 files. We found that the phenomenon of co-occurring TD is highly spread, i.e., more than 90% of Python files have been affected by at least two types of TD at the same time. In particular, from the 1722 possible associations between rules, 48 pairs frequently co-occurred in files, from which 27 are more strongly associated with each other. Among them, Design Debt issues more commonly co-occur with issues of similar nature. Test Debt issues frequently co-occur with other types, although the majority of such associations does not seem strong. Furthermore, we found that co-occurring issues tend to be introduced within the same week. However, after one of the co-occurring issues is removed, the second one survives for much longer (often around a year). Moreover, co-occurring Design Debt issues seem to be paid back in a shorter time, while most of the Test Debt issues are introduced before the other debt types but are also the last to be repaid.

RQ_{3.a}: What TD types are commonly self-fixed and how long does self-fixed TD survive?

To answer this research question, an empirical study was carried out to investigate the phenomenon of self-fixed technical debt. We analyzed the evolution of TD in 17K consecutive commits of 20 Python projects and 27K consecutive commits of 16 Java projects from the ASF. Moreover, we detected 54K Python TD issues and 71K Java TD issues based on 49 Python rules and 52 Java rules defined by SonarQube. We found that about half of the fixed TD issues were paid back by the same developers who introduced them. Moreover, the projects that are larger and have more commits and TD issues tend to have a relatively lower likelihood of observing self-fixing. We also found that there is no significant difference in the survival time of self-fixed and non-self-fixed issues. Looking at the types of TD, our findings suggest that Defect Debt and Code Debt are more likely to be self-fixed than other debt types. The results indicate that developers are more likely to self-fix TD when the issue is related to code-level aspects (e.g., compared to Design Debt and Documentation Debt). Moreover, we observed that developers who are more dedicated to or knowledgeable of the project (e.g., by contributing more commits) may drive the overall self-fixing up. Finally, the

longer the developers are involved in a project, the less often they self-fix TD.

RQ_{3.b}: Do practitioners intentionally self-fix TD and why?

For this research question, we conducted an empirical study to investigate the phenomenon of self-fixed technical debt from the perspective of software developers. In particular, we surveyed 181 practitioners spread among 17 projects (written in Java or Python) to explore their perception of this phenomenon and under what circumstances they self-fix TD. The results show that most practitioners display a positive attitude towards self-fixing TD, and seem mostly attentive to Code Debt. Moreover, when looking into the reasons to introduce and then self-fix TD, we found that non-technical reasons (e.g., related to planning and management) are the most common to justify both introducing and self-fixing TD, and are often mentioned together with technical reasons (e.g., related to the development process). In particular, many developers mention a sense of responsibility as a factor for self-fixing, and that repayment decisions are not made easily but by balancing costs and benefits, among other factors.

RQ₄: What is the process between incurring a TD item and paying it back?

To answer this research question, we conducted an empirical study that investigated the timeline of technical debt items resolved in issue trackers. Especially, we investigated the likelihood of TD items being also resolved in source code, the length between the various moments in the timeline, and how the number and involvement of developers are related to the timeline. We randomly selected 600 issues from each project, totalling 3,000 issues from the five projects. Then, we manually analyzed 200 issues that contained technical debt items and used *git log* to extract the related code commits. Finally, we classified 219 TD items from the analyzed 200 issues. We found that most of the TD items identified and mentioned as resolved in issue trackers (i.e., 88.7%) are also paid back in source code, while about one-fifth (i.e., 41/219) are discussed further after being identified. The resolved TD items in issue trackers related to Design Debt have a significantly higher chance to be paid back in source code, while Test Debt items are less likely. Furthermore, although TD items take a long time to be identified (around one year), they tend to be resolved in source code within a few days after that. Also, TD items seem to be more quickly identified in the issue tracker by the same developers who introduced them in the source code. After TD items are identified in the issue tracker, they are also more quickly resolved in the source code by the same developers who identified them. However, how developers have been involved in the issue tracker discussions does not seem to affect the length of intervals.

7.2 Future Work

Several opportunities for future work could be identified based on the work documented in this thesis. These opportunities can be grouped into three main directions: (a) extension of the research scope to other programming languages, (b) investigation of other human factors that might be related to the findings, and (c) consideration of the industrial applicability of the findings.

7.2.1 Exploration of Additional Programming Languages

The differences between TD remediation in Python and Java projects (e.g., presented and discussed in Sections 2.3 and 4.5) hint at the possibility that the language properties and the culture that emerges in language-specific software ecosystems may have an impact on the debt repayment practices. However, more research must be done to empirically understand the impact of individual language features (e.g., dynamic typing) on the studied types of debt; we can also investigate whether delaying remediation is optimal for languages that contain these features, or if there could be better strategies. Moreover, it would be interesting to find out whether the rationale for prioritizing TD remediation is related to particular programming language features or properties.

7.2.2 Human Factors

Although we have managed to analyze certain factors that are related to developers' behavior (i.e., roles, experience and contribution of practitioners), we have certainly not exhausted the wide range of human and social factors that affect software development. For example, the phenomenon of co-occurring TD might be affected by habits of developers not investigated in that study. In addition, differences in business culture (e.g., related to team dynamics) and team composition (e.g., roles and number of team members playing each role) may affect decision making about TD repayment. Finally, it would be interesting to investigate more factors related to the survival time and the likelihood of TD issues being self-fixed, such as commit frequency, commit content and code comments.

7.2.3 Industrial Applicability

Like many studies in the field of TD, we used SonarQube to conduct some of the data collection regarding TD identification and measurement. However, tools using different strategies could lead to variations in detecting and measuring the TD issues. It remains to be seen whether the use of different tools and approaches to

identify TD and its repayment can yield different results than what is reported in this thesis. Furthermore, the PhD project focused mainly on analyzing open-source projects; understanding how TD evolves in industrial ecosystems is also valuable since limited work has been done in that direction. We would particularly like to investigate whether a web dashboard that would allow developers to compare their debt remediation statistics with those of the Apache systems would be a good starting point for discussions and actionable insight.

Finally, the results from this PhD project expose several opportunities for supporting TD repayment. For example, tools could be created or updated to help developers prioritize TD remediation according to the self-fixing rate of specific projects and developers (e.g., whether they should pay the issues themselves or not), as well as the average survival time of specific TD items (e.g., what types of TD items should be repaid faster). Such tools could also provide additional insights to teams and help them assign maintenance tasks based on expected time-to-fix. Similar information can be used to warn developers and team leaders about potential issues and help them make more informed decisions prior to refactoring activities.

Appendix A

A.1 Supplementary Tables to Chapter 2

Table A.2: Project statistics

Project Name	Commits	SLOC	Classes	Age (days)	Contributors	Fixed Issues	Open Issues	Fixing Rate
Airavata-django-portal	743	41,192	614	663	5	1,277	3,225	28.37%
Airflow	6,074	89,773	1,044	1,528	742	1,387	3,935	26.06%
Allura	9,271	77,042	1,194	3,345	45	3,730	4,833	43.56%
Bloodhound	1,238	74,658	1,068	2,230	9	855	5,244	14.02%
Cassandra-dtest	5,148	61,333	487	2,619	68	3,337	3,957	45.75%
Chemistry-cmislib	117	5,020	86	2,601	3	187	215	46.52%
Cloudstack-docs-rn	197	111	0	1,653	15	1	26	3.70%
Cloudstack-documentation	59	45	1	119	16	0	2	0
Cloudstack-ec2stack	341	4,456	21	575	5	253	127	66.58%
Cloudstack-gcestack	332	3,102	37	1,789	4	116	309	27.29%
Comdev-reporter	34	1,341	3	0	1	0	161	0
Couchdb-documentation	964	944	22	2,179	105	17	63	21.25%
Fluo-muchos	261	983	4	1,416	7	37	41	47.44%
Incubator-ariatosca	190	35,603	557	634	11	557	2,467	18.42%
Incubator-milagro-mfa-server	25	4,598	84	532	7	14	401	3.37%
Incubator-mxnet	9,381	95,883	743	1,302	677	22,142	7,434	74.87%
Incubator-pagespeed-drp	40	611	10	2,597	4	1	33	2.94%
Incubator-retired-cotton	56	4,125	74	223	2	5	251	1.95%
Incubator-retired-wave-docs	10	183	0	994	3	0	10	0
Incubator-sdap-edge	2	4,900	94	19	1	0	940	0
Incubator-sdap-nexus	73	17,056	173	388	5	89	2,281	3.76%
Incubator-sdap-nexusprotp	13	48	0	350	1	0	10	0
Incubator-sdap-ningesterpy	28	1,623	45	300	3	8	155	4.91%
Incubator-senssoft-distill	98	913	12	622	5	274	73	78.96%
Incubator-senssoft-userale-pyqt5	40	612	10	525	1	142	40	78.02%
Incubator-spot	654	7,602	38	1,009	20	993	729	57.67%
Incubator-superset	3,922	16,279	90	1,205	344	39	487	7.41%
Incubator-warble-node	29	753	7	18	1	0	75	0
Infrastructure-puppet	10,942	12,112	122	1,624	60	524	1,289	28.90%
Kibble	586	8,103	8	386	4	71	354	16.71%
Kibble-scanners	152	3,940	12	362	1	8	386	2.03%
Libcloud	6,287	126,083	1,394	1,156	264	3,692	5,132	41.84%
Openwhisk-composer-python	83	1,655	50	219	5	217	244	47.07%
Openwhisk-deploy-mesos	2	1,373	4	249	2	0	93	0
Openwhisk-package-kafka	202	1,157	9	786	11	77	123	38.50%
Openwhisk-utilities	48	364	0	515	7	2	9	18.18%
Predictionio-sdk-python	122	1,487	22	1,604	9	119	93	56.13%
Qpid-dispatch	2,160	29,628	447	1,874	17	863	1,437	37.52%
Qpid-interop-test	184	4,660	59	1,156	2	217	220	49.66%
Qpid-python	1,004	19,285	385	4,232	23	352	1,482	19.19%
Steve	384	2,662	6	377	4	85	276	23.55%
Tashi	449	12,311	90	1,582	3	926	3,639	20.29%
Trafficserver-qa	109	1,083	36	724	3	51	108	32.08%
Usergrid-python	2	600	15	0	1	0	79	0
Median	187	4,032	38	1,092	5			
Average	1,410	17,667	209	1,381	57			

Table A.3: Issue rules

ID	Description	Type	Severity	Category	Issues	Rate%
3	<code>..init..</code> should not return a value	Bug	BL	Code	1	0
7	Methods and field names should not differ only by capitalization [■]	CS	BL	Code	43	34.88
12	<code>"<>"</code> should not be used to test inequality	CS	MA	Defect	2	100
13	Jump statements should not be followed by other statements	Bug	MA	Code	47	76.60
14	Docstrings should be defined	CS	MA	Document	31,364	41.48
15	Identical expressions should not be used on both sides of a binary operator [■]	Bug	MA	Code	31	19.35
16	Sections of code should not be "commented out" [■]	CS	MA	Code	1,558	55.58
20	Functions should not be too complex	CS	CR	Design	812	40.51
21	The "print" statement should not be used	CS	MA	Defect	2,396	56.34
23	<code>"\"</code> should only be used as an escape character outside of raw strings	Bug	MA	Code	1,312	68.52
24	Control flow statements should not be nested too deeply [■]	CS	CR	Design	1,265	41.98
25	Redundant pairs of parentheses should be removed [■]	CS	MA	Code	29	44.82
26	Two branches in a conditional structure should not have exactly the same implementation [■]	CS	MA	Design	320	48.75
29	Cognitive Complexity of functions should not be too high [■]	CS	CR	Design	3,250	40.89
31	The "exec" statement should not be used	Bug	BL	Defect	5	0
36	A field should not duplicate the name of its containing class [■]	CS	MA	Code	28	28.57
38	Statements should be on separate lines [■]	CS	MA	Code	1,806	53.99
40	Functions should not contain too many return statements	CS	MA	Design	706	36.40
41	Collapsible "if" statements should be merged [■]	CS	MA	Design	491	41.75
42	Function names should comply with a naming convention	CS	MA	Code	1,770	45.71
45	Nested blocks of code should not be left empty [■]	CS	MA	Code	180	37.22
46	Functions, methods and lambdas should not have too many parameters	CS	MA	Code	2,278	54.30
47	Files should not have too many lines of code [■]	CS	MA	Design	264	31.44
49	Variables should not be self-assigned [■]	Bug	MA	Defect	78	55.13
50	Lines should not be too long	CS	MA	Code	12,747	49.80
51	Python parser failure	CS	MA	Defect	19	78.95
52	Backticks should not be used	Bug	BL	Defect	5	0
54	Track uses of "FIXME" tags [■]	CS	MA	Document	173	26.59
56	Lines should have sufficient coverage by tests	CS	MA	Test	7,595	35.17
57	Source files should have a sufficient density of comment lines	CS	MA	Document	4,567	48.48
58	Source files should not have any duplicated blocks [■]	CS	MA	Design	1,346	54.38
273	Bad option value	CS	MA	Defect	22	59.09
274	Calling of not callable	CS	MA	Defect	82	79.27
276	Access of nonexistent member	CS	MA	Defect	3,473	41.41
281	Syntax error	CS	MA	Defect	972	56.89
299	Too many arguments for logging format string	CS	MA	Defect	12	25.00
328	Redefined function/class/method	CS	MA	Defect	106	57.55
355	Method has no argument	CS	MA	Defect	39	43.59
360	Undefined name	CS	MA	Defect	2,975	38.15
363	Method should have "self" as first argument	CS	MA	Defect	105	30.48
364	Format string ends in middle of conversion specifier	CS	MA	Defect	2	0
368	Not enough arguments for format string	CS	MA	Defect	8	12.50
369	Too many arguments for format string	CS	MA	Defect	38	57.89
389	Access to member before its definition	CS	MA	Defect	49	61.22
390	Method hidden by attribute of super class	CS	MA	Design	23	34.78
393	Using variable before assignment	CS	MA	Defect	5	40.00
394	Undefined variable	CS	MA	Defect	6,385	56.82
403	Too few arguments	CS	MA	Defect	198	19.70
411	Bad first argument given to super	CS	MA	Defect	31	38.71
414	Passing unexpected keyword argument in function call	CS	MA	Defect	91	56.04
418	Too many positional arguments for function call	CS	MA	Defect	128	28.12
424	NotImplemented raised - should raise NotImplementedError	CS	MA	Defect	45	82.22
432	Mixed tabs/spaces indentation	CS	MA	Defect	3,544	34.45
438	Assigning to function call which doesn't return	CS	MA	Code	3	66.67
439	Raising only allowed for classes, instances or strings	CS	MA	Defect	328	2.13
440	Bad except clauses order	CS	MA	Design	1	100

¹ ID number of the rules² Category (Code=Code Debt, Design=Design Debt, Defect=Defect Debt, Document=Documentation Debt, Test=Test Debt)[■] Also appeared in the Java study Digkas et al. (2018)

Appendix B

B.1 Supplementary Table to Chapter 5

Table B.4: Self-fixing rates for individual rules

Language	ID ^a	Definition	SR ^b	FR ^c	Issues	P ^d
Python	C438	Assigning to function call which doesn't return	1.00	1.00	1	1
Python	C25	Redundant pairs of parentheses should be removed	1.00	0.13	8	2
Python	F368	Not enough arguments for format string	1.00	1.00	1	1
Python	F412	Multiple values passed for parameter in function call	1.00	1.00	1	1
Python	C43	Related "if/else if" statements should not have the same condition	1.00	1.00	1	1
Java	F775	Package declaration should match source file directory	1.00	1.00	1557	4
Python	F393	Using variable before assignment	0.80	0.56	9	6
Python	F360	Undefined name	0.75	0.53	1675	16
Java	C422	Methods and field names should not be the same or differ only by capitalization	0.73	0.30	275	14
Python	F432	Mixed tabs/spaces indentation	0.72	0.98	54	3
Python	C13	Jump statements should not be followed by other statements	0.71	0.83	29	6
Java	F423	Anonymous inner classes containing only one method should become lambdas	0.67	0.40	718	12
Python	F49	Variables should not be self-assigned	0.67	0.27	11	5
Java	C717	Factory method injection should be used in "@Configuration" classes	0.67	0.50	6	1
Java	F677	Encryption algorithms should be used with secure mode and padding scheme	0.67	0.64	28	5
Python	F51	Python parser failure	0.65	1.00	31	9
Python	F328	Redefined function/class/method	0.60	0.78	127	11
Python	C7	Methods and field names should not differ only by capitalization	0.58	0.41	29	3
Java	F288	Asserts should not be used to check the parameters of a public method	0.58	0.18	846	7
Java	C285	Try-with-resources should be used	0.56	0.56	224	14
Python	F363	Method should have "self" as first argument	0.52	0.53	116	5

Language	ID ^a	Definition	SR ^b	FR ^c	Issues	P ^d
Python	S24	Control flow statements “if”, “for”, “while”, “try” and “with” should not be nested too deeply	0.51	0.41	499	14
Python	F281	Syntax error	0.50	0.64	817	19
Python	C45	Nested blocks of code should not be left empty	0.50	0.55	51	11
Python	S41	Collapsible “if” statements should be merged	0.49	0.54	226	15
Java	C632	Constructors should not be used to instantiate “String”, “BigInteger”, “BigDecimal” and primitive-wrapper classes	0.48	0.78	321	13
Python	C16	Sections of code should not be “commented out”	0.47	0.73	566	18
Python	C42	Function names should comply with a naming convention	0.47	0.46	321	15
Python	F21	The “print” statement should not be used	0.47	0.68	1478	18
Python	C38	Statements should be on separate lines	0.47	0.50	1282	7
Python	F418	Too many positional arguments for function call	0.46	0.24	110	6
Python	C36	A field should not duplicate the name of its containing class	0.46	0.46	28	6
Java	T425	Tests should not be ignored	0.46	0.26	360	11
Java	S573	Two branches in a conditional structure should not have exactly the same implementation	0.44	0.50	214	15
Java	F476	Raw types should not be used	0.44	0.51	8871	16
Python	F403	Too few arguments	0.43	0.93	57	9
Java	C486	Unused “private” classes should be removed	0.42	0.67	36	13
Python	F276	Access of nonexistent member	0.42	0.58	1871	19
Python	S29	Cognitive Complexity of functions should not be too high	0.41	0.50	1353	19
Java	F289	Resources should be closed	0.41	0.49	385	15
Java	S779	Cognitive Complexity of methods should not be too high	0.40	0.45	4703	16
Python	S390	Method hidden by attribute of super class	0.40	0.80	25	3
Java	C431	Sections of code should not be commented out	0.40	0.64	5444	16
Python	F394	Undefined variable	0.40	0.62	4361	18
Java	C369	Constant names should comply with a naming convention	0.39	0.43	1441	14
Java	C784	Synchronized classes Vector, Hashtable, Stack and StringBuffer should not be used	0.38	0.54	551	16
Java	C437	switch statements should have “default” clauses	0.37	0.26	596	14
Python	S26	Two branches in a conditional structure should not have exactly the same implementation	0.37	0.65	195	15
Java	C389	Standard outputs should not be used directly to log anything	0.37	0.28	2735	16
Java	F453	ThreadLocal variables should be cleaned up when no longer used	0.37	0.41	140	14
Java	C799	Unused “private” methods should be removed	0.37	0.60	472	15
Java	F750	Local variables should not shadow class fields	0.37	0.48	1404	16
Java	C386	Nested blocks of code should not be left empty	0.36	0.61	522	15
Java	C681	Unused “private” fields should be removed	0.36	0.68	793	16
Java	S337	String literals should not be duplicated	0.35	0.48	3343	16
Java	S837	Source files should not have any duplicated blocks	0.35	0.46	3121	16
Python	F424	NotImplemented raised - should raise NotImplementedError	0.35	0.96	74	4
Python	D57	Source files should have a sufficient density of comment lines	0.35	0.52	2004	20
Python	F274	Calling of not callable	0.35	0.72	115	7
Python	C46	Functions, methods and lambdas should not have too many parameters	0.35	0.75	729	12
Java	C693	Unused method parameters should be removed	0.35	0.55	2988	16
Python	S40	Functions should not contain too many return statements	0.34	0.39	344	15
Python	F355	Method has no argument	0.33	0.50	36	7

Language	ID ^a	Definition	SR ^b	FR ^c	Issues	P ^d
Python	F411	Bad first argument given to super	0.33	0.63	24	7
Java	C596	entrySet() should be iterated when both the key and value are needed	0.33	0.36	161	14
Java	T434	Thread.sleep should not be used in tests	0.33	0.13	1357	16
Java	D730	Deprecated elements should have both the annotation and the Javadoc tag	0.32	0.58	1214	16
Java	F323	Class names should not shadow interfaces or superclasses	0.32	0.26	157	11
Python	S58	Source files should not have any duplicated blocks	0.31	0.63	604	17
Java	C753	Assertion arguments should be passed in the correct order	0.31	0.40	1597	15
Java	C624	Empty arrays and collections should be returned instead of null	0.30	0.34	506	16
Java	F405	Generic exceptions should never be thrown	0.30	0.44	4695	16
Java	C333	String function use should be optimized for single characters	0.30	0.41	235	16
Java	C754	Utility classes should not have public constructors	0.29	0.23	1177	16
Java	S671	Collapsible “if” statements should be merged	0.29	0.51	1343	16
Python	D14	Docstrings should be defined	0.28	0.54	13109	20
Python	S20	Functions should not be too complex	0.28	0.53	328	13
Java	F666	Methods should not be empty	0.27	0.29	3367	16
Java	F542	Generic wildcard types should not be used in return parameters	0.27	0.47	1306	15
Python	C50	Lines should not be too long	0.26	0.85	16168	19
Java	C752	Unused type parameters should be removed	0.25	0.78	892	13
Python	C23	\ should only be used as an escape character outside of raw strings	0.25	0.86	1423	8
Java	T740	Tests should include assertions	0.24	0.27	2578	16
Python	F414	Passing unexpected keyword argument in function call	0.23	0.85	46	5
Python	C15	Identical expressions should not be used on both sides of a binary operator	0.23	0.84	31	3
Python	F369	Too many arguments for format string	0.23	0.94	33	4
Java	D601	@Override should be used on overriding and implementing methods	0.22	0.66	1419	16
Python	T56	Lines should have sufficient coverage by tests	0.21	0.37	2683	20
Python	S47	Files should not have too many lines of code	0.20	0.57	157	9
Java	F828	Boolean expressions should not be gratuitous	0.19	0.59	472	15
Python	F439	Raising only allowed for classes, instances or strings	0.18	0.10	110	3
Java	C341	Static fields should not be updated in constructors	0.18	0.51	43	9
Java	F651	Printf-style format strings should be used correctly	0.18	0.45	1391	15
Python	F389	Access to member before its definition	0.18	0.63	62	5
Java	C447	Unused assignments should be removed	0.17	0.70	2448	15
Java	F819	Overrides should match their parent class methods in synchronization	0.14	0.29	73	11
Java	F607	Exceptions should be either logged or rethrown but not both	0.13	0.10	515	10
Java	C595	Preconditions and logging arguments should not require evaluation	0.13	0.45	919	15
Java	F464	Fields in a “Serializable” class should either be transient or serializable	0.10	0.55	1083	16
Java	C758	static members should be accessed statically	0.10	0.55	164	12
Java	F391	readResolve methods should be inheritable	0.05	0.61	33	3

^a ID number of the rule and Debt Type

(C=Code Debt, S= Design Debt, D=Documentation Debt, F=Defect Debt, T=Test Debt)

^b Self-fixing Rate: the ratio is the number of self-fixed issues divided by the number of all fixed issues

^c Fixing Rate: the ratio is the number of fixed issues divided by the number of total issues

^d The number of projects that contain the issues

Bibliography

- Abbes, M., Khomh, F., Gueheneuc, Y. and Antoniol, G.: 2011, An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension, *Proceedings of the 15th European Conference Software Maintenance and Reengineering (CSMR '11)*, IEEE, Oldenburg, Germany, pp. 181–190.
DOI: 10.1109/CSMR.2011.24
- Abidi, M., Rahman, M. S., Openja, M. and Khomh, F.: 2021, Are multi-language design smells fault-prone? an empirical study, *ACM Transactions on Software Engineering and Methodology* **30**(3), 1–56.
DOI: 10.1145/3432690
- Aghajani, E., Nagy, C., Linares-Vásquez, M., Moreno, L., Bavota, G., Lanza, M. and Shepherd, D. C.: 2020, Software documentation: the practitioners' perspective, *Proceedings of the 42nd IEEE/ACM International Conference on Software Engineering (ICSE '20)*, IEEE, Seoul, Republic of Korea, pp. 590–601.
DOI: 10.1145/3377811.3380405
- Agrawal, R., Imieliński, T. and Swami, A.: 1993, Mining association rules between sets of items in large databases, *Proceedings of the 19th ACM SIGMOD International Conference on Management of Data (SIGMOD '93)*, ACM, Washington, D.C., USA, pp. 207–216.
DOI: 10.1145/170036.170072
- Aickin, M. and Gensler, H.: 1996, Adjusting for multiple testing when reporting research results: the bonferroni vs holm methods, *American journal of public health* **86**(5), 726–728.
DOI: 10.2105/ajph.86.5.726
- Alexandru, C. V., Merchante, J. J., Panichella, S., Proksch, S., Gall, H. C. and Robles, G.: 2018, On the usage of pythonic idioms, *Proceedings of the ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ACM, Boston, MA, USA, pp. 1–11.
DOI: 10.1145/3276954.3276960

- Alexandru, C. V., Panichella, S. and Gall, H. C.: 2017, Reducing redundancies in multi-revision code analysis, *Proceedings of the IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER '17)*, IEEE, Klagenfurt, Austria, pp. 148–159.
DOI: 10.1109/SANER.2017.7884617
- Alfayez, R., Alwehaibi, W., Winn, R., Venson, E. and Boehm, B.: 2020, A systematic literature review of technical debt prioritization, *Proceedings of the 3rd International Conference on Technical Debt*, pp. 1–10.
DOI: 10.1016/j.jss.2020.110827
- Alfayez, R., Behnamghader, P., Srisopha, K. and Boehm, B.: 2018, An exploratory study on the influence of developers in technical debt, *Proceedings of the First IEEE/ACM International Conference on Technical Debt (TechDebt '18)*, IEEE, Gothenburg, Sweden, pp. 1–10.
DOI: 10.1145/3194164.3194165
- Alves, N., Ribeiro, L. F., Caires, V., Mendes, T. and Spínola, R.: 2014, Towards an ontology of terms on technical debt, *Proceedings of the 6th International Workshop on Managing Technical Debt (MTD '14)*, IEEE, Victoria, BC, Canada, pp. 1–7.
DOI: 10.1109/MTD.2014.9
- Amanatidis, T., Chatzigeorgiou, A., Ampatzoglou, A. and Stamelos, I.: 2017, Who is producing more technical debt? a personalized assessment of the principal, *Proceedings of the XP2017 Scientific Workshops*, ACM, Cologne, Germany, pp. 1–8.
DOI: 10.1145/3120459.3120464
- 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.
DOI: 10.1007/s10664-020-09869-w
- 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.
DOI: 10.1016/j.infsof.2015.04.001
- Armstrong, R. A.: 2014, When to use the bonferroni correction, *Ophthalmic and Physiological Optics* **34**(5), 502–508.
DOI: 10.1111/opo.12131
- Avgeriou, P., Kruchten, P., Ozkaya, I. and Seaman, C.: 2016, Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162), *Dagstuhl Reports* **6**(4), 110–138.
DOI: 10.4230/DagRep.6.4.110
- Avgeriou, P., 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* .
DOI: 10.1109/MS.2020.3024958

- Azzeh, M., Nassif, A. B. and Minku, L. L.: 2015, An empirical evaluation of ensemble adjustment methods for analogy-based effort estimation, *Journal of Systems and Software* **103**, 36–52.
DOI: <https://doi.org/10.1016/j.jss.2015.01.028>
- Baldassarre, M. T., Lenarduzzi, V., Romano, S. and Saarimäki, N.: 2020, On the diffuseness of technical debt items and accuracy of remediation time when using sonarqube, *Information and Software Technology* **128**.
DOI: [10.1016/j.infsof.2020.106377](https://doi.org/10.1016/j.infsof.2020.106377)
- Bates, D., Sarkar, D., Bates, M. D. and Matrix, L.: 2007, The lme4 package, *R package version* **2**(1), 74.
- Bavota, G., Canfora, G., Penta, M. D., Oliveto, R. and Panichella, S.: 2013, The evolution of project inter-dependencies in a software ecosystem: The case of apache, *Proceedings of the IEEE 29th International Conference on Software Maintenance (ICSM '13)*, IEEE, Eindhoven, the Netherlands, pp. 280–289.
DOI: [10.1109/ICSM.2013.39](https://doi.org/10.1109/ICSM.2013.39)
- Bavota, G., Qusef, A., Oliveto, R., De Lucia, A. and Binkley, D.: 2012, An empirical analysis of the distribution of unit test smells and their impact on software maintenance, *Proceedings of the IEEE 28th International Conference on Software Maintenance (ICSM '12)*, IEEE, Riva del Garda, Italy, pp. 56–65.
DOI: [10.1109/ICSM.2012.6405253](https://doi.org/10.1109/ICSM.2012.6405253)
- Bavota, G., Qusef, A., Oliveto, R., De Lucia, A. and Binkley, D.: 2015, Are test smells really harmful? an empirical study, *Empirical Software Engineering* **20**(4), 1052–1094.
DOI: [10.1007/s10664-014-9313-0](https://doi.org/10.1007/s10664-014-9313-0)
- Bavota, G. and Russo, B.: 2016, A large-scale empirical study on self-admitted technical debt, *Proceedings of the IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR '16)*, IEEE, Austin, TX, USA, pp. 315–326.
DOI: [10.1145/2901739.2901742](https://doi.org/10.1145/2901739.2901742)
- Behutiye, W. N., Rodríguez, P., Oivo, M. and Tosun, A.: 2017, Analyzing the concept of technical debt in the context of agile software development: A systematic literature review, *Information and Software Technology* **82**, 139–158.
DOI: [10.1016/j.infsof.2016.10.004](https://doi.org/10.1016/j.infsof.2016.10.004)
- Bellomo, S., Nord, R. L., Ozkaya, I. and Popeck, M.: 2016, Got technical debt? surfacing elusive technical debt in issue trackers, *Proceedings of the 13th IEEE/ACM Working Conference on Mining Software Repositories (MSR '16)*, IEEE, Austin, TX, USA, pp. 327–338.
DOI: [10.1145/2901739.2901754](https://doi.org/10.1145/2901739.2901754)
- Besker, T., Martini, A. and Bosch, J.: 2017, The pricey bill of technical debt: When and by whom will it be paid?, *Proceedings of the IEEE 33th International Conference on Software Maintenance and Evolution (ICSME '17)*, IEEE, Shanghai, China, pp. 13–23.
DOI: [10.1109/ICSME.2017.42](https://doi.org/10.1109/ICSME.2017.42)

- Besker, T., Martini, A. and Bosch, J.: 2018, Technical debt cripples software developer productivity: a longitudinal study on developers' daily software development work, *Proceedings of the First International Conference on Technical Debt (TechDebt '18)*, ACM, Gothenburg, Sweden, pp. 105–114.
DOI: 10.1145/3194164.3194178
- Besker, T., Martini, A., Lokuge, R. E., Blincoe, K. and Bosch, J.: 2018, Embracing technical debt, from a startup company perspective, *Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution (ICSME '18)*, IEEE, Madrid, Spain, pp. 415–425.
DOI: 10.1109/ICSME.2018.00051
- Biaggi, A., Arcelli Fontana, F. and Roveda, R.: 2018, An architectural smells detection tool for c and c++ projects, *Proceedings of the 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA '18)*, IEEE, Prague, Czech Republic, pp. 417–420.
DOI: 10.1109/SEAA.2018.00074
- Bland, J. M. and Altman, D. G.: 2004, The logrank test, *Bmj* **328**(7447), 1073.
DOI: 10.1136/bmj.328.7447.1073
- Borges, H., Hora, A. and Valente, M. T.: 2016, Understanding the factors that impact the popularity of github repositories, *Proceedings of the 32nd IEEE International Conference on Software Maintenance and Evolution (ICSME '16)*, IEEE, Raleigh, NC, USA, pp. 334–344.
DOI: 10.1109/ICSME.2016.31
- Breslow, N.: 1970, A generalized kruskal-wallis test for comparing k samples subject to unequal patterns of censorship, *Biometrika* **57**(3), 579–594.
DOI: 10.2307/2334776
- Campbell, G. A. and Papapetrou, P. P.: 2013, *SonarQube in Action*, 1st edn, Manning Publications Co., Greenwich, CT, USA.
DOI: 10.5555/2601770
- Chatzigeorgiou, A. and Manakos, A.: 2010, Investigating the evolution of bad smells in object-oriented code, *Proceedings of the 7th International Conference on the Quality of Information and Communications Technology (QUATIC '10)*, IEEE, Porto, Portugal, pp. 106–115.
DOI: 10.1109/QUATIC.2010.16
- Chen, Z., Ma, W., Lin, W., Chen, L., Li, Y. and Xu, B.: 2018, A study on the changes of dynamic feature code when fixing bugs: Towards the benefits and costs of python dynamic features, *Science China Information Sciences* **61**.
DOI: 10.1007/s11432-017-9153-3
- Cliff, N.: 2014, *Ordinal methods for behavioral data analysis*, Psychology Press.
DOI: 10.4324/9781315806730
- Cochran, W. G.: 2007, *Sampling techniques*, John Wiley & Sons.
DOI: 10.2134/agronj1953.00021962004500120020x

- Codabux, Z. and Williams, B.: 2013, Managing technical debt: An industrial case study, *Proceedings of the 4th International Workshop on Managing Technical Debt (MTD '13)*, IEEE, San Francisco, CA, USA, pp. 8–15.
DOI: 10.1109/MTD.2013.6608672
- Cohen, J.: 1988, *Statistical Power Analysis for the Behavioral Sciences*, Lawrence Earlbaum Associates.
DOI: 10.1016/C2013-0-10517-X
- Cunningham, W.: 1992, The wycash portfolio management system, *ACM SIGPLAN OOPS Messenger* 4(2), 29–30.
DOI: 10.1145/157710.157715
- Dale, M. R. and Izurieta, C.: 2014, Impacts of design pattern decay on system quality, *Proceedings of the ACM/IEEE 8th International Symposium on Empirical Software Engineering and Measurement (ESEM '14)*, ACM, Torino, Italy, pp. 1–4.
DOI: 10.1145/2652524.2652560
- De Bleser, J., Di Nucci, D. and De Roover, C.: 2019, Assessing diffusion and perception of test smells in scala projects, *Proceedings of the IEEE/ACM 16th International Conference on Mining Software Repositories (MSR '19)*, IEEE, Montreal, QC, Canada, pp. 457–467.
DOI: 10.1109/MSR.2019.00072
- Di Nucci, D., Palomba, F., Tamburri, D. A., Serebrenik, A. and De Lucia, A.: 2018, Detecting code smells using machine learning techniques: are we there yet?, *Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER '18)*, IEEE, Campobasso, Italy, pp. 612–621.
DOI: 10.1109/SANER.2018.8330266
- Digkas, G., Chatzigeorgiou, A., Ampatzoglou, A. and Avgeriou, P.: 2020, Can clean new code reduce technical debt density, *IEEE Transactions on Software Engineering* pp. 1–1.
DOI: 10.1109/TSE.2020.3032557
- 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?, *Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER '18)*, IEEE, Campobasso, Italy, pp. 153–163.
DOI: 10.1109/SANER.2018.8330205
- Digkas, G., Lungu, M., Chatzigeorgiou, A. and Avgeriou, P.: 2017, The evolution of technical debt in the apache ecosystem, *Proceedings of the 11th European Conference on Software Architecture (ECSA '17)*, Springer, Canterbury, UK, pp. 51–66.
DOI: 10.1007/978-3-319-65831-5_4
- Ducasse, S., Rieger, M. and Demeyer, S.: 1999, A language independent approach for detecting duplicated code, *Proceedings of the 15th IEEE International Conference on Software Maintenance (ICSM '99)*, IEEE, Oxford, UK, pp. 109–118.
DOI: 10.1109/ICSM.1999.792593

- Easterbrook, S., Singer, J., anne Storey, M. and Damian, D.: 2008, Selecting empirical methods for software engineering research, *Guide to Advanced Empirical Software Engineering* pp. 285–311.
DOI: 10.1007/978-1-84800-044-5_11
- Ernst, N. A., Bellomo, S., Ozkaya, I., Nord, R. L. and Gorton, I.: 2015, Measure it? manage it? ignore it? software practitioners and technical debt, *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '15)*, ACM, Bergamo, Italy, pp. 50–60.
DOI: 10.1145/2786805.2786848
- Eyolfson, J., Tan, L. and Lam, P.: 2014, Correlations between bugginess and time-based commit characteristics, *Empirical Software Engineering* **19**(4), 1009–1039.
DOI: 10.1007/s10664-013-9245-0
- Field, A.: 2017, *Discovering statistics using IBM SPSS statistics, 5th edition*, SAGE Publications, London.
DOI: 10.5555/2502692
- Fink, A.: 2003, *The survey handbook*, SAGE Publications.
- Fischer, M., Pinzger, M. and Gall, H.: 2003, Populating a release history database from version control and bug tracking systems, *Proceeding of the 19th International Conference on Software Maintenance (ICSM '2003)*, IEEE, Amsterdam, the Netherlands, pp. 23–32.
DOI: 10.1109/ICSM.2003.1235403
- Fisher, R. A.: 1922, On the interpretation of χ^2 from contingency tables, and the calculation of p, *Journal of the Royal Statistical Society* **85**(1), 87–94.
DOI: 10.2307/2340521
- Flisar, J. and Podgorelec, V.: 2018, Enhanced feature selection using word embeddings for self-admitted technical debt identification, *Proceedings of the 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA '18)*, IEEE, Prague, Czech Republic, pp. 230–233.
DOI: 10.1109/SEAA.2018.00045
- Fontana, F. A., Ferme, V. and Zanoni, M.: 2015, Towards assessing software architecture quality by exploiting code smell relations, *Proceedings of the IEEE/ACM 2nd International Workshop on Software Architecture and Metrics (SAM '15)*, IEEE, Florence, Italy, pp. 1–7.
DOI: 10.1109/SAM.2015.8
- Fontana, F. A., Roveda, R. and Zanoni, M.: 2016, Tool support for evaluating architectural debt of an existing system: An experience report, *Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC '16)*, ACM, Pisa, Italy, pp. 1347–1349.
DOI: 10.1145/2851613.2851963
- Freire, S., Rios, N., Gutierrez, B., Torres, D., Mendonça, M., Izurieta, C., Seaman, C. and Spinola, R. O.: 2020, Surveying software practitioners on technical debt payment practices

- and reasons for not paying off debt items, *Proceedings of the 24th Evaluation and Assessment in Software Engineering (EASE '20)*, ACM, Trondheim, Norway, pp. 210–219.
DOI: 10.1145/3383219.3383241
- Fucci, G., Zampetti, F., Serebrenik, A. and Di Penta, M.: 2020, Who (self) admits technical debt?, *Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution (ICSME '20)*, IEEE, Adelaide, Australia, pp. 672–676.
DOI: 10.1109/ICSME46990.2020.00070
- Gastwirth, J. L.: 1972, The estimation of the lorenz curve and gini index, *The Review of Economics and Statistics* **54**(3), 306–16.
DOI: 10.2307/1937992
- Ghezzi, G., Würsch, M., Giger, E. and Gall, H. C.: 2012, An architectural blueprint for a plug-gable version control system for software (evolution) analysis, *Proceedings of the 2nd International Workshop on Developing Tools As Plug-Ins (TOPI '12)*, IEEE, Piscataway, NJ, USA, pp. 13–18.
DOI: 10.1109/TOPI.2012.6229803
- Gilson, F., Morales-Trujillo, M. and Mathews, M.: 2020, How junior developers deal with their technical debt?, *Proceedings of the 3rd International Conference on Technical Debt (TechDebt '20)*, ACM, Seoul, Republic of Korea, pp. 51–61.
DOI: 10.1145/3387906.3388624
- Griffith, I., Izurieta, C. and Huvaere, C.: 2017, An industry perspective to comparing the sqale and quamoco software quality models, *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '17)*, IEEE, Markham, Ontario, Canada, pp. 287–296.
DOI: 10.1109/ESEM.2017.42
- Gu, Z., Gu, L., Eils, R., Schlesner, M. and Brors, B.: 2014, Circlize implements and enhances circular visualization in r, *Bioinformatics* **30**(19), 2811–2812.
DOI: 10.1093/bioinformatics/btu393
- Guo, Y., Spínola, R. O. and Seaman, C.: 2016, Exploring the costs of technical debt management—a case study, *Empirical Software Engineering* **21**(1), 159–182.
DOI: 10.1007/s10664-014-9351-7
- Harrell, F.: 2019, Package ‘hmisc’, CRAN2018 **2019**, 235–236.
- Horgan, J. R., London, S. and Lyu, M. R.: 1994, Achieving software quality with testing coverage measures, *Computer* **27**(9), 60–69.
DOI: 10.1109/2.312032
- Iammarino, M., Zampetti, F., Aversano, L. and Di Penta, M.: 2019, Self-admitted technical debt removal and refactoring actions: Co-occurrence or more?, *Proceedings of the 35th IEEE International Conference on Software Maintenance and Evolution (ICSME '19)*, IEEE, Cleveland,

- OH, USA, pp. 186–190.
DOI: 10.1109/ICSME.2019.00029
- Iammarino, M., Zampetti, F., Aversano, L. and Di Penta, M.: 2021, An empirical study on the co-occurrence between refactoring actions and self-admitted technical debt removal, *Journal of Systems and Software* **178**, 110976.
DOI: 10.1016/J.JSS.2021.110976
- Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M. and Damian, D.: 2014, The promises and perils of mining github, *Proceedings of the 11th working conference on mining software repositories (MSR '14)*, ACM, Hyderabad, India, pp. 92–101.
DOI: 10.1145/2597073.2597074
- Kaplan, E. L. and Meier, P.: 1958, Nonparametric estimation from incomplete observations, *Journal of the American statistical association* **53**(282), 457–481.
DOI: 10.2307/2281868
- Kitchenham, B. A., Dyba, T. and Jorgensen, M.: 2004, Evidence-based software engineering, *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, IEEE Comput. Soc, Edinburgh, UK, pp. 23–28.
DOI: 10.1109/icse.2004.1317449
- Krippendorff, K.: 2011, Computing krippendorff's alpha-reliability, *Departmental Papers (ASC)* pp. 1–12.
- Krippendorff, K.: 2018, *Content analysis: An introduction to its methodology*, Sage publications.
DOI: 10.1111/j.1468-4446.2007.00153.10.x
- Kruchten, P., Nord, R. L. and Ozkaya, I.: 2012, Technical debt: From metaphor to theory and practice, *IEEE Software* **29**(6), 18–21.
DOI: 10.1109/MS.2012.167
- Lenarduzzi, V., Besker, T., Taibi, D., Martini, A. and Fontana, F. A.: 2021, A systematic literature review on technical debt prioritization: Strategies, processes, factors, and tools, *Journal of Systems and Software* **171**, 110827.
DOI: 10.1016/j.jss.2020.110827
- Lenarduzzi, V., Saarimaki, N. and Taibi, D.: 2019, On the diffuseness of code technical debt in java projects of the apache ecosystem, *Proceedings of the IEEE/ACM 2nd International Conference on Technical Debt (TechDebt '19)*, IEEE, Montreal, QC, Canada, pp. 98–107.
DOI: 10.1109/TechDebt.2019.00028
- Lenth, R. V.: 2001, Some practical guidelines for effective sample size determination, *The American Statistician* **55**(3), 187–193.
DOI: 10.1198/000313001317098149

- Lerina, A. and Nardi, L.: 2019, Investigating on the impact of software clones on technical debt, *Proceedings of the 2nd IEEE/ACM International Conference on Technical Debt (TechDebt '19)*, IEEE, Montreal, QC, Canada, pp. 108–112.
DOI: 10.1109/TechDebt.2019.00029
- 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.
DOI: 10.1007/s10664-005-1290-x
- Letouzey, J.: 2012, The sqale method for evaluating technical debt, *Proceedings of the 3rd International Workshop on Managing Technical Debt (MTD '12)*, IEEE, Zurich, Switzerland, pp. 31–36.
DOI: 10.1109/MTD.2012.6225997
- Letouzey, J. and Coq, T.: 2010, The sqale analysis model: An analysis model compliant with the representation condition for assessing the quality of software source code, *Proceedings of the 2nd International Conference on Advances in System Testing and Validation Lifecycle (VALID '10)*, Nice, France, pp. 43–48.
DOI: 10.1109/VALID.2010.31
- Letouzey, J. and Ilkiewicz, M.: 2012, Managing technical debt with the sqale method, *IEEE Software* **29**(6), 44–51.
DOI: 10.1109/MS.2012.129
- Li, Y., Soliman, M. and Avgeriou, P.: 2020, Identification and remediation of self-admitted technical debt in issue trackers, *Proceedings of the 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA '20)*, IEEE, Portorož, Slovenia, pp. 495–503.
DOI: 10.1109/SEAA51224.2020.00083
- Li, Z., Avgeriou, P. and Liang, P.: 2015, A systematic mapping study on technical debt and its management, *Journal of Systems and Software* **101**(C), 193–220.
DOI: 10.1016/j.jss.2014.12.027
- Liu, J., Huang, Q., Xia, X., Shihab, E., Lo, D. and Li, S.: 2021, An exploratory study on the introduction and removal of different types of technical debt in deep learning frameworks, *Empirical Software Engineering* **26**(2), 1–36.
DOI: 10.1007/s10664-020-09917-5
- Lozano, A. and Wermelinger, M.: 2008, Assessing the effect of clones on changeability, *Proceedings of the IEEE 24th International Conference on Software Maintenance (ICSM '08)*, IEEE, Beijing, China, pp. 227–236.
DOI: 10.1109/ICSM.2008.4658071
- Lu, C.: 2019, But do commit messages matter? an empirical association analysis with technical debt, *Joint Proceedings of the Summer School on Software Maintenance and Evolution*, Tampere, Finland, pp. 45–53.

- Lu, Y., Mao, X., Li, Z., Zhang, Y., Wang, T. and Yin, G.: 2016, Does the role matter? an investigation of the code quality of casual contributors in github, *Proceedings of the 23rd Asia-Pacific Software Engineering Conference (APSEC '16)*, IEEE, Hamilton, New Zealand, pp. 49–56.
DOI: 10.1109/APSEC.2016.018
- Ma, W., Chen, L., Zhang, X., Zhou, Y. and Xu, B.: 2017, How do developers fix cross-project correlated bugs?: A case study on the github scientific python ecosystem, *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*, IEEE, Piscataway, NJ, USA, pp. 381–392.
DOI: 10.1109/ICSE.2017.42
- Maldonado, E. D. S., Abdalkareem, R., Shihab, E. and Serebrenik, A.: 2017, An empirical study on the removal of self-admitted technical debt, *Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME '17)*, IEEE, Shanghai, China, pp. 238–248.
DOI: 10.1109/ICSME.2017.8
- Maldonado, E. D. S. and Shihab, E.: 2015, Detecting and quantifying different types of self-admitted technical debt, *Proceedings of the IEEE 7th International Workshop on Managing Technical Debt (MTD '15)*, IEEE, Bremen, Germany, pp. 9–15.
DOI: 10.1109/MTD.2015.7332619
- March, S. T. and Smith, G. F.: 1995, Design and natural science research on information technology, *Decision Support Systems* **15**(4), 251–266.
DOI: 10.1016/0167-9236(94)00041-2
- Marcilio, D., Bonifácio, R., Monteiro, E., Canedo, E., Luz, W. and Pinto, G.: 2019, Are static analysis violations really fixed?: A closer look at realistic usage of sonarqube, *Proceedings of the 27th International Conference on Program Comprehension (ICPC '19)*, IEEE, Montreal, Quebec, Canada, pp. 209–219.
DOI: 10.1109/ICPC.2019.00040
- Marinescu, R.: 2012, Assessing technical debt by identifying design flaws in software systems, *IBM Journal of Research and Development* **56**(5), 9:1–9:13.
DOI: 10.1147/JRD.2012.2204512
- 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.
DOI: 10.1016/j.infsof.2015.07.005
- McCulloch, C. E. and Neuhaus, J. M.: 2005, Generalized linear mixed models, *Encyclopedia of biostatistics* **4**.
DOI: 10.1002/0470011815.b2a10021

- Miles, M. B. and Huberman, A. M.: 1994, *Qualitative data analysis: An expanded sourcebook*, SAGE Publications.
DOI: 10.1016/s0272-4944(05)80231-2
- Mittas, N. and Angelis, L.: 2013, Ranking and clustering software cost estimation models through a multiple comparisons algorithm, *IEEE Transactions on Software Engineering* **39**(4), 537–551.
DOI: 10.1109/TSE.2012.45
- Moore, D. S.: 2018, *The Basic Practice of Statistics*, 8th edn, W.H. Freeman & Company.
- 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, *Proceedings of the ACM/IEEE 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM '09)*, IEEE, Lake Buena Vista, FL, USA, pp. 390–400.
DOI: 10.1109/ESEM.2009.5314231
- Oliveira, E., Fernandes, E., Steinmacher, I., Cristo, M., Conte, T. and Garcia, A.: 2020, Code and commit metrics of developer productivity: a study on team leaders perceptions, *Empirical Software Engineering* **25**(4), 2519–2549.
DOI: 10.1007/s10664-020-09820-z
- Palomba, F., Bavota, G., Penta, M. D., Fasano, F., Oliveto, R. and Lucia, A. D.: 2018a, A large-scale empirical study on the lifecycle of code smell co-occurrences, *Information & Software Technology* **99**, 1–10.
DOI: 10.1016/j.infsof.2018.02.004
- Palomba, F., Bavota, G., Penta, M. D., Fasano, F., Oliveto, R. and Lucia, A. D.: 2018b, On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation, *Empirical Software Engineering* **23**(3), 1188–1221.
DOI: 10.1145/3180155.3182532
- Palomba, F., Di Nucci, D., Panichella, A., Oliveto, R. and De Lucia, A.: 2016, On the diffusion of test smells in automatically generated test code: An empirical study, *Proceedings of the 9th International Workshop on Search-Based Software Testing (SBST '16)*, ACM, Austin, TX, USA, pp. 5–14.
DOI: 10.1109/SBST.2016.010
- Pérez, B., Castellanos, C., Correal, D., Rios, N., Freire, S., Spínola, R., Seaman, C. and Izurieta, C.: 2021, Technical debt payment and prevention through the lenses of software architects, *Information and Software Technology* **140**, 106692.
DOI: 10.1016/j.infsof.2021.106692
- Porting Python 2 Code to Python 3*: 2017, <https://docs.python.org/3/howto/pyporting.html>. Accessed: 2020-07-14.

- Potdar, A. and Shihab, E.: 2014, An exploratory study on self-admitted technical debt, *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME '14)*, IEEE, Victoria, British Columbia, Canada, pp. 91–100.
DOI: 10.1109/ICSME.2014.31
- Rahman, F., Bird, C. and Devanbu, P.: 2012, Clones: What is that smell?, *Empirical Software Engineering* **17**(4), 503–530.
DOI: 10.1109/MSR.2010.5463343
- Rahman, F. and Devanbu, P.: 2011, Ownership, experience and defects: a fine-grained study of authorship, *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*, IEEE, Honolulu, HI, USA, pp. 491–500.
DOI: 10.1145/1985793.1985860
- Ray, B., Posnett, D., Filkov, V. and Devanbu, P.: 2014, A large scale study of programming languages and code quality in github, *Proceedings of the ACM SIGSOFT 22nd International Symposium on Foundations of Software Engineering (FSE '14)*, Hong Kong, China, pp. 155–165.
DOI: 10.1145/2635868.2635922
- Rios, N., Spínola, R. O., Mendonça, M. and Seaman, C.: 2020, The practitioners' point of view on the concept of technical debt and its causes and consequences: a design for a global family of industrial surveys and its first results from brazil, *Empirical Software Engineering* **25**(5), 3216–3287.
DOI: 10.1007/s10664-020-09832-9
- 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 ACM/IEEE 12th International Symposium on Empirical Software Engineering and Measurement (ESEM '18)*, ACM, Oulu, Finland, pp. 1–10.
DOI: 10.1145/3239235.3268917
- Romano, J., Kromrey, J. D., Coraggio, J. and Skowronek, J.: 2006, Appropriate statistics for ordinal level data: Should we really be using t-test and cohen'sd for evaluating group differences on the nsse and other surveys, *Annual Meeting of the Florida Association of Institutional Research*, Vol. 177.
- Rosner, B., Glynn, R. J. and Lee, M.-L. T.: 2006, The wilcoxon signed rank test for paired comparisons of clustered data, *Biometrics* **62**(1), 185–192.
DOI: 10.1111/j.1541-0420.2005.00389.x
- Runeson, P., Host, M., Rainer, A. and Regnell, B.: 2012, *Case Study Research in Software Engineering: Guidelines and Examples*, Wiley Blackwell.
DOI: 10.5555/2361717
- Salamea, M. J. and Farré, C.: 2019, Influence of developer factors on code quality: a data study, *Proceedings of the 19th IEEE International Conference on Software Quality, Reliability and*

- Security Companion (QRS-C '19)*, IEEE, Sofia, Bulgaria, pp. 120–125.
DOI: 10.1109/QRS-C.2019.00035
- Scott, A. J. and Knott, M.: 1974, A cluster analysis method for grouping means in the analysis of variance, *Biometrics* pp. 507–512.
DOI: 10.2307/2529204
- Seaman, C.: 1999, Qualitative methods in empirical studies of software engineering, *IEEE Transactions on software engineering* **25**(4), 557–572.
DOI: 10.1109/32.799955
- Shamshiri, S., Rojas, J. M., Galeotti, J. P., Walkinshaw, N. and Fraser, G.: 2018, How do automatically generated unit tests influence software maintenance?, *Proceedings of the IEEE 11th International Conference on Software Testing, Verification and Validation (ICST '18)*, IEEE, Västerås, Sweden, pp. 250–261.
DOI: 10.1109/ICST.2018.00033
- Sharma, T., Fragkoulis, M. and Spinellis, D.: 2017, House of cards: Code smells in open-source c# repositories, *Proceedings of the ACM/IEEE 11th International Symposium on Empirical Software Engineering and Measurement (ESEM '17)*, IEEE, Toronto, ON, Canada, pp. 424–429.
DOI: 10.1109/ESEM.2017.57
- Sharma, T., Singh, P. and Spinellis, D.: 2020, An empirical investigation on the relationship between design and architecture smells, *Empirical Software Engineering* **25**(5), 4020–4068.
DOI: doi 10.1007/s10664-020-09847-2
- Shull, F., Singer, J. and Sjøberg, D. I.: 2007, *Guide to advanced empirical software engineering*, Springer.
DOI: 10.5555/1951869
- Siegel, S. and Castellan, N.: 1988, *Nonparametric statistics for the behavioral sciences*, 2nd edn, McGraw-hill.
DOI: 10.1126/science.126.3267.266.a
- Sjøberg, D. I. K., Yamashita, A., Anda, B. C. D., Mockus, A. and Dybå, T.: 2013, Quantifying the effect of code smells on maintenance effort, *IEEE Transactions on Software Engineering* **39**(8), 1144–1156.
DOI: 10.1109/TSE.2012.89
- Spadini, D., Palomba, F., Zaidman, A., Bruntink, M. and Bacchelli, A.: 2018, On the relation of test smells to software code quality, *Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution (ICSME '18)*, IEEE, Madrid, Spain, pp. 1–12.
DOI: 10.1109/ICSME.2018.00010
- Steinmacher, I., Treude, C. and Gerosa, M. A.: 2019, Let me in: Guidelines for the successful onboarding of newcomers to open source projects, *IEEE Software* **36**(4), 41–49.
DOI: 10.1109/MS.2018.110162131

- Tan, J., Feitosa, D. and Avgeriou, P.: 2020, An empirical study on self-fixed technical debt, *Proceedings of the 3rd IEEE/ACM International Conference on Technical Debt (TechDebt '20)*, ACM, pp. 1–12.
DOI: 10.1145/3387906.3388621
- Tan, J., Feitosa, D., Avgeriou, P. and Lungu, M.: 2021, Evolution of technical debt remediation in python: A case study on the apache software ecosystem, *Journal of Software: Evolution and Process* **33**(4), e2319.
DOI: 10.1002/smr.2319
- Tantithamthavorn, C., McIntosh, S., Hassan, A. E. and Matsumoto, K.: 2016, An empirical comparison of model validation techniques for defect prediction models, *IEEE Transactions on Software Engineering* **43**(1), 1–18.
DOI: 10.1109/TSE.2016.2584050
- Tom, E., Aurum, A. and Vidgen, R.: 2013, An exploration of technical debt, *Journal of Systems and Software* **86**(6), 1498–1516.
DOI: 10.1016/j.jss.2012.12.052
- Tratt, L.: 2009, Dynamically typed languages, *Advances in Computers* **77**, 149–184.
DOI: 10.1016/S0065-2458(09)01205-4
- Trendowicz, A. and Münch, J.: 2009, Factors influencing software development productivity-state-of-the-art and industrial experiences, *Advances in computers* **77**, 185–241.
DOI: 10.1016/S0065-2458(09)01206-6
- Tsay, J., Dabbish, L. and Herbsleb, J.: 2014, Influence of social and technical factors for evaluating contribution in github, *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*, ACM, Hyderabad, India, pp. 356–366.
DOI: 10.1145/2568225.2568315
- Tsoukalas, D., Kehagias, D., Siavvas, M. and Chatzigeorgiou, A.: 2020, Technical debt forecasting: an empirical study on open-source repositories, *Journal of Systems and Software* **170**, 110777.
DOI: 10.1016/j.jss.2020.110777
- 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 IEEE/ACM 31th International Conference on Automated Software Engineering (ASE '16)*, ACM, Singapore, Singapore, pp. 4–15.
DOI: 10.1145/2970276.2970340
- Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Penta, M. D., 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.
DOI: 10.1109/TSE.2017.2653105

- Vadlamani, S. L. and Baysal, O.: 2020, Studying software developer expertise and contributions in stack overflow and github, *Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution (ICSME '20)*, IEEE, Adelaide, Australia, pp. 312–323.
DOI: 10.1109/ICSME46990.2020.00038
- Van Solingen, R., Basili, V., Caldiera, G. and Rombach, H. D.: 2002, Goal Question Metric (GQM) approach, *Encyclopedia of Software Engineering*, John Wiley & Sons, Inc., Hoboken, NJ, USA, pp. 528–532.
DOI: 10.1002/0471028959
- Wan, Z., Xia, X., Hassan, A. E., Lo, D., Yin, J. and Yang, X.: 2018, Perceptions, expectations, and challenges in defect prediction, *IEEE Transactions on Software Engineering* **46**(11), 1241–1266.
DOI: 10.1109/TSE.2018.2877678
- Wang, B., Chen, L., Ma, W., Chen, Z. and Xu, B.: 2015, An empirical study on the impact of python dynamic features on change-proneness, *Proceedings of the 27th International Conference on Software Engineering and Knowledge Engineering (SEKE '15)*, KSI Research Inc., Pittsburgh, USA, pp. 134–139.
DOI: 10.18293/SEKE2015-097
- Wehaibi, S., Shihab, E. and Guerrouj, L.: 2016, Examining the impact of self-admitted technical debt on software quality, *Proceedings of the 23rd IEEE international conference on software analysis, evolution, and reengineering (SANER '16)*, Vol. 1, IEEE, Osaka, Japan, pp. 179–188.
DOI: 10.1109/SANER.2016.72
- Wen, F., Nagy, C., Lanza, M. and Bavota, G.: 2020, An empirical study of quick remedy commits, *Proceedings of the 28th International Conference on Program Comprehension (ICPC '20)*, ACM, Seoul, Republic of Korea, pp. 60–71.
DOI: 10.1145/3387904.3389266
- Wieringa, R.: 2010, Design science methodology: principles and practice, *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 2, pp. 493–494.
DOI: 10.1145/1810295.1810446
- Wieringa, R.: 2014, *Design Science Methodology for Information Systems and Software Engineering*, Springer Berlin Heidelberg, Berlin, Heidelberg.
DOI: 10.1007/978-3-662-43839-8
- Xavier, L., Ferreira, F., Brito, R. and Valente, M. T.: 2020, Beyond the code: Mining self-admitted technical debt in issue tracker systems, *Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20)*, pp. 137–146.
DOI: 10.1145/3379597.3387459
- Xia, X., Lo, D., Shihab, E., Wang, X. and Zhou, B.: 2015, Automatic, high accuracy prediction of reopened bugs, *Automated Software Engineering* **22**(1), 75–109.
DOI: 10.1007/s10515-014-0162-2

- Yamashita, A. and Moonen, L.: 2013, Exploring the impact of inter-smell relations on software maintainability: An empirical study, *Proceedings of the 35th International Conference on Software Engineering (ICSE '13)*, pp. 682–691.
DOI: 10.1109/ICSE.2013.6606614
- Yamashita, A., Zanoni, M., Fontana, F. A. and Walter, B.: 2015, Inter-smell relations in industrial and open source systems: A replication and comparative analysis, *Proceedings of the IEEE 31st International Conference on Software Maintenance and Evolution (ICSME '15)*, IEEE, Bremen, Germany, pp. 121–130.
DOI: 10.1109/ICSM.2015.7332458
- Yan, M., Xia, X., Shihab, E., Lo, D., Yin, J. and Yang, X.: 2018, Automating change-level self-admitted technical debt determination, *IEEE Transactions on Software Engineering* **45**(12), 1211–1229.
DOI: 10.1109/TSE.2018.2831232
- Zampetti, F., Serebrenik, A. and Di Penta, M.: 2018, Was self-admitted technical debt removal a real removal? an in-depth perspective, *Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18)*, ACM, Gothenburg, Sweden, pp. 526–536.
DOI: 10.1145/3196398.3196423
- Zampetti, F., Serebrenik, A. and Di Penta, M.: 2020, Automatically learning patterns for self-admitted technical debt removal, *Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '20)*, IEEE, London, ON, Canada, pp. 355–366.
DOI: 10.1109/SANER48275.2020.9054868
- Zar, J. H.: 1972, Significance testing of the spearman rank correlation coefficient, *Journal of the American Statistical Association* **67**(339), 578–580.
DOI: 10.2307/2284441
- Zhai, H., Casalnuovo, C. and Devanbu, P.: 2019, Test coverage in python programs, *Proceedings of the IEEE/ACM 16th International Conference on Mining Software Repositories (MSR '19)*, ACM, Montreal, QC, Canada, pp. 116–120.
DOI: 10.1109/MSR.2019.00027
- Zou, W., Lo, D., Chen, Z., Xia, X., Feng, Y. and Xu, B.: 2018, How practitioners perceive automated bug report management techniques, *IEEE Transactions on Software Engineering* **46**(8), 836–862.
DOI: 10.1109/TSE.2018.2870414