UNIVERSITY OF MACEDONIA
SCHOOL OF INFORMATION SCIENCES
DEPARTMENT OF APPLIED INFORMATICS

A study on the evolution of software quality and technical debt in open source applications

Theodoros Amanatidis

PhD Dissertation

Supervisor: Prof. Alexander Chatzigeorgiou
Advisors: Prof. Maria Satratzemi, Prof. Ioannis Stamelos

Thessaloniki, 05/2020

"A study on the evolution of software quality and technical debt in open source applications"

Theodoros Amanatidis

BSc Financial Accounting 2011, MSc Applied Informatics 2014

PhD Dissertation

A dissertation submitted in fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Macedonia

Thessaloniki, Greece

2020

This dissertation was submitted during the massive pandemic of the

corona virus COVID-19

with over 4M cases and 280K deaths all over the world

at the time of this submission

Summary

Almost all software systems around us evolve constantly to accommodate new requirements, to adapt in changing environments and to fix known issues. Software evolution analysis can reveal important information concerning maintenance practices followed by development teams. The goal of this dissertation is to study the evolution of open source web applications by investigating the evolution of software quality and maintenance cost as captured by the metaphor of Technical Debt (TD). Technical Debt is a concept in programming which reflects the extra maintenance effort that arises when “sub-optimal” code, that is easy to implement in the short run, is used instead of applying best practices.

This research work investigates the evolution of a large number of open source web applications. The applications were analyzed in the context of Lehman’s eight (8) laws of software evolution and particularly, it has been examined whether the laws are confirmed in practice for open source web applications. Lehman’s Laws are, Continuing Change (Law I), Increasing Complexity (Law II), Self-Regulation (Law III), Conservation of Organizational Stability (Law IV), Conservation of Familiarity (Law V), Continuing Growth (Law VI) Declining Quality (VII) and Feedback System (VIII). The results provide evidence that the evolution of web applications comply with most of the laws.

The TD metaphor has received increasing attention from the research community in the last years. Particularly, a large portion of research studies have focused on the notion of TD Interest, which reflects the additional maintenance effort that is incurred due to the existence of sub-optimal software (i.e., due to the existence of Technical Debt). Adopting the state-of-research on TD Management, this dissertation has aimed at investigating the impact of TD on corrective maintenance and particularly, to what extent does the presence of TD in software modules slows down development pace by increase the time and effort required for fixing bugs.

Although TD is usually assessed on either the entire system or on individual software modules and most studies focus on the identified inefficiencies, it is the actual craftsmanship of developers that causes the accumulation of TD. Driven by the fact that TD is introduced by the developers themselves, this dissertation attempts to explore the relation between developers’ characteristics and the tendency to introduce inefficiencies that lead to TD.

Despite the fact that TD is an established and recognized concept in the software engineering community, it also remains a metaphor and like all metaphors it is inherently abstract. This means that the way it is defined and interpreted by software engineering stakeholders constitutes a subjective matter. Each software professional has developed his/her own perception around TD prioritization and management. Thus, many voices question the validity of the way it is detected and calculated by automated static analysis tools. To this end, in the context of this research a survey has been sent to a large number of developers that contribute to open source web applications in order to gain insights into the factors that lead developers to adopt or reject fixes as suggested by automated static code analysis.

By acknowledging the lack of a ground truth regarding the assessment of TD among the existing TD tools, this dissertation attempts to extract a set of classes with high TD, as detected by three (3) major TD tools (CAST AIP, Squore and SonarQube). These classes can serve as benchmark of validated high-TD classes and this way a basis can be established that can be used either for prioritization of maintenance activities or for training more sophisticated TD identification techniques. To this end, the current research work proposes a methodology to extract a benchmark set of validated high-TD classes while at the same time, it reveals three types of class profiles that successfully capture the spectrum of TD measurements provided by the three TD tools.

Finally, the overall contribution of this dissertation comprises five (5) points; (a) it provides valuable undiscovered information on how open source web applications evolve over time since web-based systems had received limited attention in contrast to desktop ones, (b) considering the lack of empirical evidence on the relation between TD amount and TD Interest, it investigates to what extent the presence of TD in software modules slows down development pace by increasing the time and effort required for fixing bugs, (c) by acknowledging that it is the actual craftsmanship of the developers that causes the accumulation of TD, this work outlines the characteristics of the developers who tend to add TD in open source applications, (d) it sheds light into the reasons that drive developers to agree or disagree with automatically detected TD whose urgency is very often questionable by developers and (e) by acknowledging the lack of a basis regarding the detection and prioritization of TD among the existing TD tools, the current work proposes a methodology to extract a benchmark set of modules which are ranked as high-TD modules by three (3) TD tools altogether.

Keywords: software evolution, Lehman's laws, software quality, software maintenance, open-source applications, software repositories, technical debt, technical debt management, corrective maintenance, archetypal analysis, inter-rater agreement, ground truth, benchmark

Contents

Chapter I. INTRODUCTION 1

1. Software Evolution and Technical Debt 1
2. Dissertation Goals and Research Questions 1
   - 2.1. Evolution of Web Applications 4
   - 2.2. Technical Debt and Corrective Maintenance 4
   - 2.3. Personalized Assessment of Technical Debt Principal 5
   - 2.4. Factors Affecting Decision to Repay Technical Debt 5
   - 2.5. Benchmark of Technical Debt Liabilities 6
3. Dissertation outline 6

Chapter II. BACKGROUND ON TECHNICAL DEBT 7

1. Foreword 7
2. Key Components of TD 7
3. What TD really is 8
4. TD Types 9
5. Activities and Strategies for Managing TD 10
6. Tools assessing TD (TD tools) 11
7. A comment on Technical Debt Management 13

Chapter III. EVOLUTION OF WEB APPLICATIONS 16

1. Introduction 16
2. Related Work 18
3. Case Study Design 19
   - 3.1. Goal and Research Question 19
   - 3.2. Selection of Cases 20
   - 3.3. Employed Process and Tools 22
   - 3.4. Data Analysis 23
4. Results and Discussion 27
   - 4.1. Law I: Continuing Change 27
   - 4.2. Law II: Increasing Complexity 29
   - 4.3. Law III: Self-Regulation 30
   - 4.4. Law IV: Conservation of Organizational Stability 32
   - 4.5. Law V: Conservation of Familiarity 34
   - 4.6. Law VI: Continuing Growth 36

4.7. Law VII: Declining Quality ... 37
4.8. Law VIII: Feedback System ... 41
5. Overview and Comparison to Previous Work ... 43
5.1. Summary of Results ... 43
5.2. Comparison to Previous Work ... 44
6. Implications for Researchers and Practitioners ... 46
7. Threats to Validity ... 47
8. Conclusions ... 48
Chapter IV. TECHNICAL DEBT AND CORRECTIVE MAINTENANCE ... 51
1. Introduction ... 51
2. Case Study Design ... 52
2.1. Goal and Research Questions ... 52
2.2. Cases and Units of Analysis ... 53
2.3. Data Collection ... 54
2.4. Data Analysis ... 55
3. Results ... 55
4. Threats to Validity ... 56
5. Discussion and Conclusions ... 57
Chapter V. PERSONALIZED ASSESSMENT OF TECHNICAL DEBT PRINCIPAL ... 59
1. Introduction ... 59
2. Related Work ... 60
3. Case Study Design ... 61
3.1. Research Objectives and Research Questions ... 61
3.2. Case and Units of Analysis ... 61
3.3. Variables and Data Collection ... 62
3.3.1. Variables ... 62
3.3.2. Data Collection ... 63
3.4. Data Analysis ... 63
4. Results and Discussion ... 64
4.1. Distribution of TD among Developers ... 64
4.2. TD Violations per Developer ... 66
4.3. TD vs. Developer Maturity ... 67
5. Implications of the Study ... 68

6. Threats to Validity ... 68
7. Conclusions ... 69
Chapter VI. FACTORS AFFECTING DECISION TO REPAY TECHNICAL DEBT ... 71
1. Introduction ... 71
2. Related Work ... 72
3. Study Design ... 72
3.1. Personalized report to participants ... 73
3.2. Set-up of the Study ... 74
4. Results and Discussion ... 75
4.1. Statistical Analysis ... 75
4.2. Discussion of the Results ... 77
5. Threats to Validity ... 78
6. Conclusions ... 78
Chapter VII. BENCHMARK OF TECHNICAL DEBT LIABILITIES ... 80
1. Introduction ... 80
2. TD Assessment Tools ... 83
3. Case Study Design ... 88
3.1. Goal and Research Questions ... 88
3.2. Selection of Cases ... 89
3.3. Data Collection ... 89
3.4. Data Analysis Methodology ... 92
3.4.1. Inter-rater Agreement (RQ_1) ... 92
3.4.2. Benchmarking through Archetypal Analysis ($RQ_2 – RQ_4$) ... 94
4. Results and Discussion ... 102
4.1. RQ_1: To what extent do the assessors (tools) agree in the ranking of classes in terms of TD measurement? ... 102
4.2. RQ_2: How many archetypes (reference assessments) are required to capture the diversity of the tools? ... 105
4.3. RQ_3: Which are the characteristics of the extracted archetypes? ... 106
4.4. RQ_4: Which classes should be selected to form an agreement-based benchmark of top-TD modules? ... 108
5. Implications to Practitioners and Researchers ... 111
6. Threats to Validity ... 112
7. Related Work ... 113

7.1. Comparison of Tools measuring Technical Debt 114
7.2. Benchmarks in Software Maintenance 115
8. Conclusions and Future Work 116
Chapter VIII. CONCLUSIONS AND FUTURE WORK 122
1. Conclusions and Contribution 122
1.1. Evolution of Web Applications 122
1.2. Technical Debt and Corrective Maintenance 123
1.3. Personalized Assessment of Technical Debt Principal 124
1.4. Factors Affecting Decision to Repay Technical Debt 124
1.5. Benchmark of Technical Debt Liabilities 125
2. Future Work 127
Publications 129
1. Journals 129
2. Conferences 129
References 130

List of Figures

Chapter II. BACKGROUND ON TECHNICAL DEBT 7

Figure 1. Trend of worldwide web search for topic "Technical Debt" (January 2004 – March 2020) 7

Figure 2. Extra effort to add functionality (TD Interest) due to existence of TD [6] 8

Chapter III. EVOLUTION OF WEB APPLICATIONS 16

Figure 1. (a) File and (b) function breakdown of examined projects based on their latest release . 22

Figure 2. Workflow for analyzing types and frequency of changes in PHP projects 22

Figure 3. Calculation of incremental growth, maintenance effort and growth rate (example) 25

Figure 4. Trend of Days Between Releases metric for project usebb 28

Figure 5. Evolution of Days Between Releases metric for projects with p-value > 0.05 29

Figure 6. Trendlines of CCN/LOC for projects with p-value < 0.05 30

Figure 7. Ripples in the total number of functions/methods for phpMyFAQ 31

Figure 8. Evolution of Incremental Growth for projects with p-value > 0.05 32

Figure 9. Evolution of Maintenance Effort ($V_{4.1}$) for projects with p-value > 0.05 34

Figure 10. Evolution of Number of Commits ($V_{4.2}$) for projects with p-value > 0.05 34

Figure 11. Evolution of Incremental Changes for projects with p-value > 0.05 36

Figure 12. Trendlines of LOC for projects with p-value < 0.05 37

Figure 13. Examination of the validity of the 8th law in project "*mustache*" 42

Chapter IV. TECHNICAL DEBT AND CORRECTIVE MAINTENANCE 51

Figure 1. Corrective maintenance at file level 54

Figure 2. Discriminative power of TD amount (left/right bars correspond to low/high TD files, respectively) 56

Chapter V. PERSONALIZED ASSESSMENT OF TECHNICAL DEBT PRINCIPAL 59

Figure 1. Process of obtaining TD deltas for each developer 63

Figure 2. Distribution of TD among developers 65

Figure 3. TD violation types per developer 66

Figure 4. Introduced TD versus developer maturity 67

Chapter VI. FACTORS AFFECTING DECISION TO REPAY TECHNICAL DEBT 71

Figure 1. Anonymized TD report of a developer in Yii2 73

Figure 2. Evaluation screen for a TD item in project Yii2 74

Chapter VII. BENCHMARK OF TECHNICAL DEBT LIABILITIES 80

Figure 1. Shortcomings from diverse TD measurements 81

Figure 2. Scatter plot for rankings of TD measurements of opennlp project as evaluated by TD tools (CAST, Squore) 93

Figure 3. Archetypal solutions (CAST, Squore) for opennnlp project 96
Figure 4. RSS plot (CAST, Squore) for opennnlp project) 97
Figure 5. Reference assessment profiles (archetypes) (opennnlp project) 98
Figure 6. Scatter plot for neighboring classes to the Max-Ruler archetype (CAST, Squore) (opennnlp project) 99
Figure 7. Dot plots with the aggregated results of Kendall's W concordance coefficient 102
Figure 8. Box plots (a) and error bars (b) of the distributions of Kendall's W concordance coefficient 103
Figure 9. Scatter plot (3D) for the rankings of the TD assessments (all three tools) (opennlp project) 105
Figure 10. RSS plot (SonarQube, CAST, Squore) (opennnlp project) 106
Figure 11. Reference assessment profiles (archetypes) from the assessments by all three tools (opennnlp project) 107
Figure 12. Percentage of top-rated classes assessed by all three tools for increasing levels of threshold values a (sensitivity analysis) 109
Figure 13. Post-hoc analysis for LME model (sensitivity analysis) 110

List of Tables

Chapter I. INTRODUCTION 1

Table 1. Timeline of the dissertation's research (driving stimulus and publication output for each goal) 2

Chapter III. EVOLUTION OF WEB APPLICATIONS 16

Table 1. Top-ten Languages of Public Open Source Projects Hosted By SourceForge & Github 17

Table 2. Most Updated Formulation of Lehman's Laws 18

Table 3. Overview of Examined Projects 20

Table 4. Data Analysis 23

Table 5. Correlation Between Variables 26

Table 6. Statistical Results on Law I (Continuing Change) 28

Table 7. Statistical Results on Law II (Increasing Complexity) 30

Table 8. Statistical Results on Law III (Self-Regulation) 31

Table 9. Statistical Results on Law IV (Conservation of Organizational Stability) 33

Table 10. Statistical Results on Law V (Conservation of Familiarity) 35

Table 11. Statistical Results on Law VI (Continuing Growth) 37

Table 12. Statistical Results on Law VII (Declining Quality) 39

Table 12. (continued) Statistical Results on Law VII (Declining Quality) 40

Table 13. Statistical Results on Law VIII (feedback system) 43

Table 14. Summary of findings about Lehman's laws 44

Table 15. Primary Measures Employed for the Investigation of Laws in Previous Studies 45

Table 16. Validity of Lehman's Laws According to Various Studies 46

Chapter IV. TECHNICAL DEBT AND CORRECTIVE MAINTENANCE 51

Table 1. Analyzed Projects 54

Table 2. Data Analysis 55

Table 3. Spearman's Correlation Results 56

Chapter V. PERSONALIZED ASSESSMENT OF TECHNICAL DEBT PRINCIPAL 59

Table 1. OSS PHP Project Demographics 62

Table 2. Data Analysis 64

Chapter VI. FACTORS AFFECTING DECISION TO REPAY TECHNICAL DEBT 71

Table 1. Frequency distributions for categorical variables 75

Table 2. Descriptive statistics for continuous variables 75

Table 3. Parameters of the final model 77

Chapter VII. BENCHMARK OF TECHNICAL DEBT LIABILITIES 80
Table 1. List of identified TD assessment tools 86
Table 2. List of TD tools with the conditions that they satisfied for their inclusion 88
Table 3. Characteristics of analyzed projects 91
Table 4. Representation of the dataset from the TD assessment results from each employed tool 92
Table 5. Indicative set of classes that are close to the Max-Ruler archetype (CAST, Squore) (opennnlp project) 100
Table 6. Kendall's W Concordance Coefficient among all three TD tools for each analyzed system 104
Table 7. Estimated mean percentage with 95% CI for each threshold value α (sensitivity analysis) 110

Chapter I. INTRODUCTION

1. Software Evolution and Technical Debt

Software Evolution constitutes part of the broader area of Software Engineering. The main goal of software evolution research is to study and understand the way software systems evolve over time so as to predict and prevent future inefficiencies. This can ease software maintenance and consequently reduce maintenance effort and cost.

The domain of software evolution exists since the 70's when M. Lehman formulated the laws of software evolution (more details in Chapter III). Two of his laws refer to *"increasing complexity"* and *"declining quality"*. Specifically, it was suggested that the complexity of a system will increase and its quality will decline over time, hindering software maintenance, if no proactive measures are taken. The degradation of software quality that Lehman talked about in the 70's is captured, nowadays, by the Technical Debt (TD) metaphor.

Technical Debt is a concept in programming that highly affects software maintenance and, consequently, software evolution. It reflects the extra maintenance effort that arises when "non-optimal" code, that is easy to implement in the short run, is used instead of applying best practices.

The distance between the optimal state of the software and the actual one can be considered as the **principal** of the software's Technical Debt, in analogy to the principal of Financial Debt (more details in Chapter II). To eliminate (or repay) TD principal means that actions have to be taken (e.g. refactogings or even code rewrite) in order to eliminate the distance from the optimal state of the software.

The aforementioned additional effort that has to be spent on software maintenance due to the existence of TD is considered as the **interest** of software's TD, just like in Financial Debt (more details in Chapter II). As TD accumulates during software evolution, software maintenance becomes more and more complex and, consequently, more expensive in terms of time and cost. The excessive accumulation of TD can be a serious threat for any software system. For this reason, continuous quantification and monitoring of TD should be of high priority.

To this end, the purpose of this dissertation is to study the evolution of open source applications by investigating evolution of software quality and technical debt. The objective of this dissertation can be decomposed into five more specific goals which are described in the next section.

2. Dissertation Goals and Research Questions

This section presents the five goals of this dissertation, in summary. The studies that have been conducted to achieve each goal are developed in a dedicated chapter. Before reading the following five goals, the reader can find the timeline of the research conducted during this dissertation in Table 1. The timeline contains the chronicle of dissertation's goals along with the driving stimulus for each goal and the resulting publication.

Table 1. Timeline of the dissertation's research (driving stimulus and publication output for each goal)

Timeline	Stimulus	Goal	Publication
Jun 2014 – Nov 2015	Lehman's Laws	Investigate whether Lehman's Laws of software evolution are confirmed in practice for open source web applications *(Chapter III)*	(Amanatidis and Chatzigeorgiou, 2016)[1]
Nov 2015 – Oct 2016	Increasing Interest in Technical Debt since 2012	Investigate the impact of a module's TD on its corrective maintenance (i.e., bug fixing) *(Chapter IV)*	(Amanatidis et al., 2017a)[2]
Oct 2016 – May 2017	Technical Debt comes from developers themselves	Explore the relation between developers' characteristics and the tendency to evoke violations that lead to TD *(Chapter V)*	(Amanatidis et al., 2017b)[3]

[1] Amanatidis, Theodoros & Chatzigeorgiou, Alexander. (2016). Studying the Evolution of PHP Web Applications. Information and Software Technology. 72. 48-67. 10.1016/j.infsof.2015.11.009.
[2] Amanatidis, Theodoros & Chatzigeorgiou, Alexander & Ampatzoglou, Apostolos. (2017). The Relation between Technical Debt and Corrective Maintenance in PHP Web Applications. Information and Software Technology. 87. 10.1016/j.infsof.2017.05.004.
[3] Amanatidis, Theodoros & Chatzigeorgiou, Alexander & Ampatzoglou, Apostolos & Stamelos, Ioannis. (2017). Who is Producing More Technical Debt? A Personalized Assessment of TD Principal. Proceedings of the 9th International Workshop on Managing Technical Debt (MTD' 17), Cologne, Germany.

May 2017

May 2018

Mar 2020

Stimulus	Goal	Publication
Really urgent to resolve?	Shed light into the factors that lead developers to adopt or reject fixes as suggested by automated static code analysis *(Chapter VI)*	(Amanatidis et al., 2018)[4]
Which TD tool to employ?	Propose methodology to extract benchmark set of validated high-TD modules *(Chapter VII)*	(Amanatidis et al., 2020)[5] *submitted for publication and revised

[4]Amanatidis, Theodoros & Mittas, Nikolaos & Chatzigeorgiou, Alexander & Ampatzoglou, Apostolos & Angelis, Lefteris. (2018). The developer's dilemma: Factors affecting the Decision to Repay Code Debt. Proceedings of the 2018 International Conference on Technical Debt (TechDEBT), Gothenburg, Sweden.

[5]Amanatidis, T., Mittas, N., Moschou, A., Chatzigeorgiou, A., Ampatzoglou, A., Angelis, L. (2020). Evaluating the Agreement among Technical Debt Measurement Tools: Building an Empirical Benchmark of Technical Debt Liabilities. *Submitted for publication (and revised) to the Empirical Software Engineering Journal (EMSE)*.

2.1. Evolution of Web Applications

Software evolution analysis can reveal important information concerning maintenance practices. Most of the studies which analyze software evolution focus on desktop applications written in compiled languages, such as Java and C. However, a vast amount of the web content today is powered by web applications written in PHP and thus the evolution of software systems written in such a scripting language deserves a distinct analysis.

To obtain an overview of the way open-source systems evolve over time, the goal of this part of dissertation is to analyze the evolution of open-source PHP projects in an attempt to investigate whether Lehman's laws of software evolution are confirmed in practice for web applications. To achieve this goal, data (changes and metrics) have been collected for successive versions of thirty (30) PHP projects while statistical tests (primarily trend tests) have been employed to evaluate the validity of each law on the examined web applications. Based on the aforementioned first goal of the dissertation, the following research question has been formulated:

RQ: *Is the evolution of web applications written in PHP compliant with Lehman's laws of evolution*?

The research question can be decomposed into eight research questions, one for each of Lehman's laws. The detailed work on the Evolution of Open Source Web Applications is presented in Chapter III.

2.2. Technical Debt and Corrective Maintenance

After establishing whether open-source software systems comply with most of Lehman's laws of evolution, the dissertation has focused on the impact of Technical Debt on software evolution. Considering that corrective maintenance consumes a significant part of developers' activity over the evolution of any software project it becomes interesting to investigate whether the presence of inefficiencies slows down development pace by increase the time and effort required for fixing bugs.

Software teams are often asked to deliver new features within strict deadlines leading developers to deliberately or inadvertently serve "*not quite right code*" compromising software quality and maintainability. This non-ideal state of software causes the accumulation of Technical Debt (TD) which adds additional maintenance effort (i.e. interest, as aforementioned in the previous section). The objective of this part of work is to quantify how TD affects software maintenance.

Although the relation between debt amount and interest is well-defined in traditional economics (i.e., interest is proportional to the amount of debt), this relation has not yet been explored in the context of TD. To this end, one aim of this dissertation is to investigate the relation between the amount of TD and the interest that has to be paid during corrective maintenance.

To explore this relation, a case study on ten (10) open source PHP projects has been performed. The obtained data have been analyzed to assess the relation between the amount of TD and two aspects of interest: (a) corrective maintenance (i.e., bug fixing) frequency, which translates to *interest probability* and (b) corrective maintenance effort which is related to *interest amount* (see Chapter IV for details).

The goal of this part of dissertation is to examine whether the frequency and the effort spent on corrective maintenance activities of a specific module, is related to the amount of its TD. Based on this goal, the research questions can be formulated as follows:

RQ_1: *Is the TD amount of a file related to the number of times that it underwent corrective maintenance?*
RQ_2: *Is the TD amount of a file related to the extent of modification that it underwent during corrective maintenance?*

The detailed work on the Relation Between Technical Debt and Corrective Maintenance is presented in Chapter IV.

2.3. Personalized Assessment of Technical Debt Principal

Contemporary software development is assisted by a number of sophisticated tools, like Integrated Development Environments with auto-complete functionality, automation testing environments, advanced build tools that handle external dependencies, Continuous Integration pipelines that automate routine activities. Moreover, software professionals have access to enormous amounts of information in open knowledge communities (like StackOverflow) and the ability to reuse code from thousands of open-source projects, limiting the need for reinventing the wheel. Nevertheless, software programming remains to a large extent a human-centric activity and requires both experience and knowledge. Lack of expertise usually results in reduced productivity (i.e. longer development times for a given set of functionalities) or in lower software quality.

Most studies in the literature assess TD on either the entire software system or on individual software artifacts, that is, by looking at the final product of software development. Considering that it is the actual craftsmanship of developers that causes the accumulation of TD and in the light of extremely high maintenance cost, efficient software project management cannot occur without recognizing the relation between developer characteristics and the tendency to evoke violations that lead to TD.

To this end, this dissertation investigates three research questions related to the distribution of TD among the developers of a software project, the types of violations caused by each developer and the relation between developers' maturity and the tendency to accumulate TD.

RQ_1**:** *Is TD uniformly distributed among the developers of a software project?*
RQ_2**:** *Which TD violations are introduced by the developers of a software project?*
RQ_3**:** *What is the relation between TD and the maturity of developers in a software project?*

The study has been performed on four (4) widely employed PHP open-source projects. All developers' personal characteristics have been anonymized. The detailed work on the Personalized Assessment of TD Principal is presented in Chapter V.

2.4. Factors Affecting Decision to Repay Technical Debt

Although TD is an established and recognized concept in the software engineering community, it also remains a metaphor and like all metaphors it is inherently abstract. This means that the way it is defined and interpreted by software engineering stakeholders constitutes a subjective matter. Software developers often disagree with an automatically generated list of improvement suggestions, which they consider not fitting or important for their own code. To shed light into the reasons that drive developers to adopt or reject refactoring suggestions (i.e. TD repayment), this dissertation investigates the potential factors that affect the developers' decision to agree (or disagree) with the removal of a specific TD liability.

Developers of four (4) well-known open-source applications have been asked to evaluate the urgency of automatically detected code violations in the source code that they contribute. To increase the response rate, a personalized assessment has first been sent to each developer, summarizing his/her own contribution to the TD of the corresponding project. Responds have been collected through a custom-built web application that presented code fragments suffering from violations as identified by TD-assessment tool along with information that could possibly affect their level of agreement to the importance of resolving an issue.

Multivariate statistical analysis methods have been used to understand the importance and the underlying relationships among these factors and the results are expected to be useful for researchers and practitioners in TD Management. The detailed work on the Investigation of the factors that lead developers to repay TD is presented in Chapter VI.

2.5. Benchmark of Technical Debt Liabilities

Although several tools are available for assessing TD with most notable examples SonarQube, Squore and CAST AIP, each tool essentially checks software against a particular ruleset. The use of different rulesets can often be beneficial as it leads to the identification of a wider set of problems; however, for the common usage scenario where developers or researchers rely on a single tool, the diverse estimates of TD and the identification of different mitigation actions limits the credibility and applicability of the findings.

The goal of this part of dissertation is two-fold: First, to evaluate the degree of agreement between leading TD assessment tools. Second, to propose a methodology to capture the diversity of the examined tools with the aim of identifying a benchmark set of classes with respect to their level of TD (e.g., that of high TD levels in all employed tools). This way a basis can be established that can be used either for prioritization of maintenance activities or for training more sophisticated TD identification techniques. The proposed methodology is illustrated through a case study on fifty (50) open source systems employing three leading TD tools.

Based on the last goal of this dissertation, the following research questions (RQ) have been formulated:

RQ_1: *To what extent do the assessors (tools) agree in the ranking of classes in terms of TD measurement*?

RQ_2: *How many archetypes (reference assessments) are required to capture the diversity of the tools*?

RQ_3: *Which are the characteristics of the extracted archetypes*?

RQ_4: *Which classes should be selected to form an agreement-based benchmark of top-TD modules?*

The detailed work on the Proposed Methodology to Extract Benchmark Set of Validated high-TD Modules is presented in Chapter VII.

3. Dissertation outline

The rest of the dissertation is organized as follows:

Chapter II provides background information on TD to familiarize the reader with the analysis on TD in the next chapters. Particularly, Chapter II focuses on key components of TD (TD Item, TD Principal, TD Interest), various TD Types, activities and strategies for managing TD and existing tools for TD assessment (TD tools).

In Chapter III through Chapter VII the work to achieve the five goals of this dissertation is developed. Particularly:

- Chapter III investigates whether Lehman's Laws of software evolution are confirmed in practice for open source web applications.
- In Chapter IV the relation between TD and corrective maintenance (i.e., bug fixing) is investigated.
- The research work presented in Chapter V explores the relation between developers' characteristics and the tendency to evoke violations that lead to TD.
- Chapter VI attempts to shed light into the factors that lead developers to adopt or reject fixes as suggested by automated static code analysis.
- Chapter VII investigates the agreement among three (3) major TD tools and proposes a methodology to extract benchmark set of validated high-TD modules.

Finally, in Chapter VIII conclusions and contribution are presented along with suggested future work.

Chapter II. BACKGROUND ON TECHNICAL DEBT

In this chapter, basic concepts of Technical Debt (TD) and Technical Debt Management (TDM) are introduced to familiarize the reader with the analysis on TD that follows in the next chapters. Particularly, this chapter briefly discusses key components of TD (TD Item, TD Principal, TD Interest), various definitions of TD, TD Types, activities and strategies for managing TD as well as existing TD assessing tools (TD tools).

1. Foreword

Technical Debt is a metaphor in programming that was originally coined by Ward Cunningham [1]. It reflects the extra maintenance effort that arises when "dirty" code, that is easy to implement in the short run, is used instead of applying best practices.

According to Google Trends[6] (see Figure 1), the topic of "Technical Debt" has gained increasing attention in web search since 2004 which adds extra value to the contribution of this dissertation.

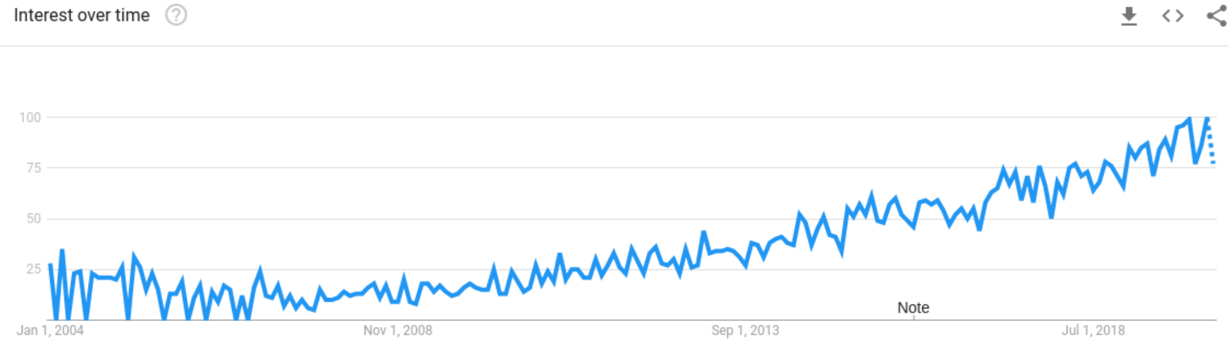


Figure 1. Trend of worldwide web search for topic "Technical Debt" (January 2004 – March 2020)

Technical Debt undermines software's maintainability and agility, rendering the addition of new functionality more and more difficult as software matures. Consequently, software evolution becomes more expensive for product owners as they are called to spend for money to address the increasing clients' needs. Possibly, these are the clients that had been won by sacrificing software quality (i.e. tolerance on TD existence) to achieve timely placement in the market.

2. Key Components of TD

There are some key components that altogether form the concept of Technical Debt. These are "TD Item", "TD Principal" and "TD Interest" and are widely discussed in the rest of the dissertation.

TD Item: The term "TD Item" refers to a violation of coding rules [3] and is considered an instance of TD [2], [4]. Each TD Item might contain information, such as the location it was detected (i.e. class or file), the estimated effort (or time) to resolve, the responsible developer, etc.

TD Principal: The term "TD Principal" refers to the estimated effort that is needed to resolve all the TD Items of the system [2], [4]. The existence of TD Items causes the software to diverge from its optimal state. The distance between the optimal state of the software and the actual one can also be considered as the principal of the software's Technical Debt (TD Principal), in analogy to the principal of Financial Debt [4], [5]. To eliminate (or repay) TD principal means that mitigation actions have to

[6] https://trends.google.com/

be taken (e.g. refactor or even rewrite part of code base) in order to minimize the distance from the optimal state of the software.

TD Interest: The term "TD Interest" refers to the aforementioned extra effort that has to be put for the addition of new functionality due to the existence of TD [4], [5]. This extra effort can be considered as the interest of software's TD (see Figure 2), just like in Financial Debt, where the borrower pays an amount of interest on top of every loan installment.

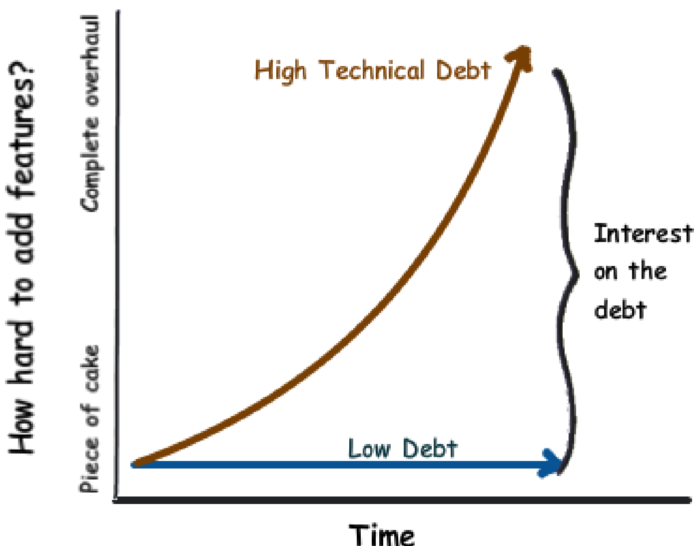


Figure 2. Extra effort to add functionality (TD Interest) due to existence of TD [6]

3. What TD really is

Just like all metaphors, TD is inherently abstract and its definition depends on personal/subjective interpretation. Researchers and practitioners have developed various definitions for it, based on their own point of view. According to Li et al. [2], stakeholders define TD based on five (5) dimensions of TD.

- TD as a metaphor to financial debt (TD Interest, TD Principal)
- Properties of TD (type, severity, effort to remediate, etc.)
- Causes of TD
- Effect of TD
- Uncertainty around TD (i.e. risk, but also opportunities that it can offer)

Some practitioners focus on the consequences of TD in their attempt to define it. They view TD as a codebase that loses agility as project matures [7]. Others, focus on the interest that has to be paid in the long run as a form of a more expensive software maintenance in terms of effort and time [8]. An academic definition of TD embraces the aspect of consequences of TD, as well [9].

"Technical debt describes the consequences of software development actions that intentionally or unintentionally prioritize client value and/or project constraints such as delivery deadlines, over more technical implementation and design considerations"

Another definition that was formulated during the Dagstuhl Seminar [10] considers TD as a collection of expedient design or implementation constructs that affect future changes:

"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."

On the other hand, some people treat *TD as a tool*. Through the lens of a startup company [11], TD can be used as a tool for getting ahead in the market. In the case of a startup company the opportunities that TD can offer might be more important than the risk that it carries.

4. TD Types

Technical Debt Types refer to specific categories of TD (e.g., architectural, design, code) or sub-categories based on the cause of TD (e.g., architectural TD can be caused by architecture smells). There are ten (10) different TD types identified in the literature [2] and each TD type represent different kind of software inefficiencies that need to be remediated in order to approach software's optimal state:

- Requirements TD: refers to lack of proper requirements specification [12].
- Architectural TD: refers to several quality issues that arise due to incorrect architecture decisions.
- Design TD: refers to technical compromises that are allowed in detailed software design.
- Code TD: involves violation of coding rules and/or coding standards
- Test TD: refers to shortcuts taken in code testing (i.e. lack of proper unit or end to end, testing)
- Build TD: usually reflects overly complex build process.
- Documentation TD: refers to incomplete or outdated documentation of software which can undermine software understandability.
- Infrastructure TD: refers to non-ideal configuration of development environments which can negatively affect the ability of the development team to bring software in production mode.
- Versioning TD: involves problems regarding source code versioning, such as misleading tagging of releases.
- Defect TD: refers to defects/bugs found in the way software operates.

In another study [13], the authors have identified five (5) more TD types in the literature on top of the aforementioned. These are:

- People TD: refers to developer problems that can delay software development. For example, when specific expertise is concentrated among few people in the development team meaning that everything has to pass from their hands.
- Test Automation TD: it is considered as a sub-category of test TD and it refers to the effort involved in automating tests to support continuous integration [14].
- Process TD: involves outdated processes, i.e. processes that no longer are needed for what they were designed in the first place [14].
- Service TD: refers to improper selection and implementation of web services that result in the divergence of the software's features from the predefined requirements.
- Usability TD: refers to inappropriate usability decisions that will have to be reconsidered later [15].

5. Activities and Strategies for Managing TD

As TD accumulates during software evolution, software maintenance becomes more and more complex and, consequently, more expensive in terms of time and cost. The excessive accumulation of TD can be a serious threat for a software. For this reason, activities for managing and conserving TD at viable level should be of high priority.

TDM is composed of several activities [2] and strategies [13] that prevent accumulation of TD and keep existing TD within reasonable levels.

The most notable examples of activities for managing TD are:

- TD Identification: refers to the detection of TD caused by technical decisions in a software system. TD detection can be conducted through specific processes, such as static code analysis.
- TD Measurement: involves the quantification of the detected TD in the system or the estimation of the overall TD of the system.
- TD Prioritization: involves the ranking of the identified TD based on predefined rules to support decision making on which TD Items to remediate first.
- TD Prevention: involves proactive measures that aim at preventing potential TD from being incurred.
- TD Monitoring: refers to the tracking of the evolution of the cost of unresolved TD in the software system.
- TD Repayment: involves mitigation actions (e.g. refactorings, reengineering, etc.) for the resolution of TD.
- TD Representation/Documentation: comprises representation and reporting of TD in a uniform manner that efficiently addresses stakeholders' concerns.
- TD Communication: involves the notification of managerial stakeholders regarding the existence of TD in the software system.

Strategies for managing TD involve combination of actions and techniques from the aforementioned TDM activities. The most notable examples of TDM strategies are:

- Cost-Benefit Analysis [5], [16]: It involves decision on when it is imperative to repay TD Principal, i.e., when the level of TD Interest exceeds a predefined threshold.
- Portfolio of TD Items [17]: The idea of this strategy is to list all detected TD Items and decide which are urgent to resolve immediately and which to postpone.
- Options [16]: The Options strategy involves the investment in the option that will facilitate improvement of the source code in the future, rather than repaying TD Principal now. The profits of this strategy are not immediate.
- Analytic Hierarchy Process [16]: This strategy involves the prioritization and ranking of TD Items based on their severity and the potential profits of their mitigation.
- Calculation of TD Principal [18]: This strategy deals with the estimation of TD Principal (based on an defined process, e.g. the OMG Specification on Automated Technical Debt Measurement [3]) and the mapping of identified TD Items with quality attributes.
- Marking of Dependencies and Code Issues [19]: This strategy involves the tagging of specific parts of code base with TD in a way that is easy to visualize and drive decision making regarding paying the TD of those code parts.

6. Tools assessing TD (TD tools)

During the previous years, numerous TD assessment tools have emerged; these tools are able to measure TD either in terms of cost or effort/time to repay TD. In this dissertation a non-systematic literature search, including grey literature (such as websites), has been conducted to locate existing TD tools. Right below, a short description of the identified TD assessment tools is provided (in alphabetical order).

AnaConDebt [20] is a tool that focuses on Architectural Debt. Since a change in the architecture of a project can be really expensive and time consuming it is important to decide if and when this change should be implemented. The tool uses a large list of internal and external factors to estimate more accurately the future principal and interest. It helps managers to decide when it is the right time to refactor the code of their software.

CAST [18] contains several sub-tools in order to provide the entire quality profile for the project. Health dashboard, Engineering dashboard, Security dashboard, CAST Appmark which is a benchmarking base to use as a comparison standard and CAST Enlighten with Imaging system that offers a visualization of the project. This tool helps companies to perform "Shift Left" techniques to detect the issues of a project in early stages of its life cycle. This way the cost of fixing the issues is more tolerable. The tool implements the C-CPP, CISQ, CWE, NIST-SP-800-53R4, OMG-ASCQM, OWASP, PCI-DSS-V3.2.1 and STIG-V4R8 standards. By performing static analysis, a list of issues is created. Only a part of the problems will be solved and this part de-fines the technical debt metric.

CodeScene [21] serves as a mean to preserve the quality of the code of the automated tests. It combines repository mining with static code analysis and machine learning. Static analysis can detect the problems in the project, but since the source code is treated as of the same importance, repository mining is necessary to recognize behavioral data and social factors that can affect future decisions of refactoring. The results of the metrics may have different meaning depending on the characteristics of each project. Machine learning is used to identify patterns in order to prioritize these metrics and assign them the appropriate weight. The final result of the tool is a catalogue with the problematic files ranked by their total impact.

DebtFlag [19] is a tool for capturing, tracking and resolving technical debt in Java systems. It consists of two parts; one plug in for Eclipse IDE which is responsible to collect the data from the source code, and one web application to visualize the results. These two applications connect via a database. The collected data is structured using the TDMF form, which was extended to cover the tool's needs. The tool offers the results in such a way that can be used to manage technical debt in two levels; project level and implementation level with micromanagement.

Debtgrep [22] is an inhouse tool developed by Ericsson 4G 5G Baseband and its purpose is to pre-vent technical debt. It uses a file where all rules are declared using regex. The rules can contain forbidden words to restrict the usage of API and deprecated methods and also guidelines for design and architectural rules. The rules can be applied only to a specific part of code such as new code. This tool supports the communication between the developing team members and enhance the consistency and the uniformity of the project.

DV8 [23] is a commercial extension of Titan [24]. DV8 functions with DRSpaces [25], which are groups of system’s files that are architecturally related. Within DRSpaces, DV8 computes three modularity metrics (Decoupling level, Propagation Cost and Independence Level) and detects six architecture anti-patterns (Clique, Package Cycle, Improper Inheritance, Unstable Interface, Crossing and Modularity Violation). DRSpaces (i.e. the subsets of architecturally related files) that are involved in a selected set of issues are called ‘architecture roots’. The tool calculates the added maintenance cost due to each instance of each anti-pattern, and the added maintenance cost of each architecture root. The source code analysis is performed by the Understand tool [26].

Kiuwan [27] is a proprietary code analysis tool that supports numerous programming languages and is capable of integrating with several IDEs. It can be obtained under a commercial license and it can also be tested within a free trial period.

NDepend [28] is a static analysis tool for .NET projects available in Visual Studio Market Place. It offers a variety of code quality metrics and a visualization of the dependencies in the project. The tool handles the source code as a form of database, and the user can define new evaluation rules using LINQ to perform queries on it. Other features of the tool include reporting service and the ability of comparison between the generations of the same project.

SonarQube [29] is a widely known tool used to track the quality and maintainability of source code. The tool implements the MISRA, CWE, SANS and CERT rule standards to pro-vide measurements regarding complexity, duplications, code issues, maintainability, quality gates in combination with technical debt, reliability, security, project size and test coverage. In addition, there are many plugins to extend the available utilities, such as WebDriver for Selenium test analysis or AEM Rules set for Adobe. The measurement of technical debt is an important component of SonarQube. The tool calculates the debt by multiplying the number issues of each type with the average time the specific issue type needs to be fixed. Then the time is multiplied with the cost for each man-day. The average time and the cost can be configured by the user. It uses the SQALE method and provides a technical debt pyramid to help making decisions prioritizing tasks.

Squore [30] consists of three smaller tools. The first one, the analyzer, is used to collect data from different sources (source code, tests and hardware component information) and build the project's hierarchy tree. Then a more detailed measurement takes place for each one of the nodes based on the ISO, HIS, SPICE and MISRA rule standards. Last but not least, the tool also offers a dashboard for the visualization of the results. The tool can be a part of Jenkins continuous integration and can also recognize which files are most important to have Unit Tests in order to improve the efficiency.

TD-Tracker [31] is a web application, which provides a structured way to create a catalogue with the issues in a project. The protocol, which is implemented, consists of three stages. For the first stage there is a data collector where the problems are identified and a list is populated. The input data can come from either an external source where, with appropriate mapping, the data can be stored directly to the database of the application, or the integration with GitHub. After finishing the collection, the second stage begins where a semi-automated task takes place. A user has to review the previous list with the issues, and decide which of them are actual problems that need to be solved. Then there is the third stage with the longest duration of all three. In this stage a user assigns tasks related to technical debt and also monitors the progress of them.

TEDMA [32] is an open tool, which analyzes different indices related to technical debt during the evolution of a project. It is open to integrate with third party tools to extend the analysis. It consists of three layers. The first is called Data Layer and holds the processes used to gather information about the project, which is examined. Currently, Git repositories are used as data input. The second is the Service Layer where there are three basic services. (i) Data loader service is responsible for offering the source code in a processable form to the tool. Then analyzers such as PMD and Findbugs detect code smells and problems. (ii) Statistics service uses R to perform statistical analysis of the data. The analysis is performed at file level but it can be extended to other levels of abstraction. (iii) Technical debt management model service uses models in Java and R to support decision-making. The last layer is the Presentation Layer which is responsible for documentation and visualization.

VisminerTD [33] is an open source web tool which monitors and manages technical debt comparing the results between different project's versions. When an issue is detected it can be tracked to determine whether its TD was paid off or not. It uses the Repository Miner tool to collect data and metrics from code repositories. VisminerTD uses queries to the database of the Repository Miner to gather the

preferred information and present them to the user via a friendly interface. A set of graphical views are available to setup the search settings and then manage the technical debt items.

7. A comment on Technical Debt Management

Technical Debt can be either intentional or unintentional [34]. Intentional debt can serve as leverage for future growth as it can speed up productivity and allow faster presence in the market. Nevertheless, it is an imperative need to always monitor and maintain it within viable levels. On the other hand, unintentional debt cannot be acknowledged in the first place. Thus, development teams are advised to employ specialized TD tools for identification and quantification of TD Principal. However, the "blind" resolution of the all suggested fixes is neither productive nor efficient and thus, not suggested. The development teams should consider one or some of the aforementioned strategies for managing TD and prioritize the issues that are more urgent to remediate.

References

[1] W. Cunningham, "The WyCash Portfolio Management System," in *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum)*, New York, NY, USA, 1992, pp. 29–30, doi: 10.1145/157709.157715.

[2] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *J. Syst. Softw.*, vol. 101, pp. 193–220, Mar. 2015, doi: 10.1016/j.jss.2014.12.027.

[3] "About the Automated Technical Debt Measure Specification Version 1.0." https://www.omg.org/spec/ATDM/About-ATDM (accessed Mar. 13, 2020).

[4] A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou, "The financial aspect of managing technical debt: A systematic literature review," *Inf. Softw. Technol.*, vol. 64, pp. 52–73, Aug. 2015, doi: 10.1016/j.infsof.2015.04.001.

[5] A. Chatzigeorgiou, A. Ampatzoglou, A. Ampatzoglou, and T. Amanatidis, "Estimating the breaking point for technical debt," in *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, Oct. 2015, pp. 53–56, doi: 10.1109/MTD.2015.7332625.

[6] "techDebt1.png (402×397)." http://commadot.com/wp-content/uploads/2011/02/techDebt1.png (accessed Mar. 31, 2020).

[7] "Why the way we look at technical debt is wrong," *Think Big*, Feb. 27, 2017. https://www.bigeng.io/why-the-way-we-look-at-technical-debt-is-wrong/ (accessed Mar. 31, 2020).

[8] "The Fallacy of Technical Debt | Hacker Noon." https://hackernoon.com/the-fallacy-of-technical-debt-202f7406337e (accessed Mar. 31, 2020).

[9] J. Holvitie *et al.*, "Technical debt and agile software development practices and processes: An industry practitioner survey," *Inf. Softw. Technol.*, vol. 96, pp. 141–160, Apr. 2018, doi: 10.1016/j.infsof.2017.11.015.

[10] "Schloss Dagstuhl : Seminar Homepage." https://www.dagstuhl.de/en/program/calendar/semhp/?semnr=16162 (accessed Mar. 31, 2020).

[11] "What is technical debt? And why does almost every startup have it?," *freeCodeCamp.org*, Oct. 16, 2017. https://www.freecodecamp.org/news/what-is-technical-debt-and-why-do-most-startups-have-it-9a54458daabf/ (accessed Mar. 31, 2020).

[12] N. A. Ernst, "On the role of requirements in understanding and managing technical debt," in *2012 Third International Workshop on Managing Technical Debt (MTD)*, Jun. 2012, pp. 61–64, doi: 10.1109/MTD.2012.6226002.

[13] N. S. R. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman, "Identification and management of technical debt: A systematic mapping study," *Inf. Softw. Technol.*, vol. 70, pp. 100–121, Feb. 2016, doi: 10.1016/j.infsof.2015.10.008.

[14] Z. Codabux and B. Williams, "Managing technical debt: An industrial case study," in *2013 4th International Workshop on Managing Technical Debt (MTD)*, May 2013, pp. 8–15, doi: 10.1109/MTD.2013.6608672.
[15] N. Zazworka, R. O. Spínola, A. Vetro', F. Shull, and C. Seaman, "A case study on effectively identifying technical debt," in *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, Porto de Galinhas, Brazil, Apr. 2013, pp. 42–47, doi: 10.1145/2460999.2461005.
[16] C. Seaman *et al.*, "Using technical debt data in decision making: Potential decision approaches," in *2012 Third International Workshop on Managing Technical Debt (MTD)*, Jun. 2012, pp. 45–48, doi: 10.1109/MTD.2012.6225999.
[17] Y. Guo and C. Seaman, "A portfolio approach to technical debt management," in *Proceedings of the 2nd Workshop on Managing Technical Debt*, Waikiki, Honolulu, HI, USA, May 2011, pp. 31–34, doi: 10.1145/1985362.1985370.
[18] B. Curtis, J. Sappidi, and A. Szynkarski, "Estimating the Principal of an Application's Technical Debt," *IEEE Softw.*, vol. 29, no. 6, pp. 34–42, Nov. 2012, doi: 10.1109/MS.2012.156.
[19] J. Holvitie and V. Leppänen, "DebtFlag: Technical Debt Management with a Development Environment Integrated Tool," in *Proceedings of the 4th International Workshop on Managing Technical Debt*, Piscataway, NJ, USA, 2013, pp. 20–27, Accessed: May 29, 2019. [Online]. Available: http://dl.acm.org/citation.cfm?id=2663297.2663301.
[20] A. Martini and J. Bosch, "An Empirically Developed Method to Aid Decisions on Architectural Technical Debt Refactoring: AnaConDebt," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, May 2016, pp. 31–40.
[21] A. Tornhill, "Assessing Technical Debt in Automated Tests with CodeScene," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Apr. 2018, pp. 122–125, doi: 10.1109/ICSTW.2018.00039.
[22] S. Arvedahl, "Introducing Debtgrep, a Tool for Fighting Technical Debt in Base Station Software," in *Proceedings of the 2018 International Conference on Technical Debt*, New York, NY, USA, 2018, pp. 51–52, doi: 10.1145/3194164.3194183.
[23] M. Nayebi *et al.*, "A Longitudinal Study of Identifying and Paying Down Architecture Debt," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, May 2019, pp. 171–180, doi: 10.1109/ICSE-SEIP.2019.00026.
[24] L. Xiao, Y. Cai, and R. Kazman, "Titan: a toolset that connects software architecture with quality analysis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Hong Kong, China, Nov. 2014, pp. 763–766, doi: 10.1145/2635868.2661677.
[25] L. Xiao, Y. Cai, and R. Kazman, "Design rule spaces: a new form of architecture insight," in *Proceedings of the 36th International Conference on Software Engineering*, Hyderabad, India, May 2014, pp. 967–977, doi: 10.1145/2568225.2568241.
[26] "SciTools.com." https://scitools.com/ (accessed Mar. 31, 2020).
[27] "Kiuwan - End-to-end application security," *Kiuwan*. https://www.kiuwan.com/ (accessed Mar. 31, 2020).
[28] K. Chopra and M. Sachdeva, "EVALUATION OF SOFTWARE METRICS FOR SOFTWARE PROJECTS," *Int. J. Comput. Technol.*, vol. 14, no. 6, pp. 5845–5853, Apr. 2015, doi: 10.24297/ijct.v14i6.1915.
[29] G. A. Campbell and P. P. Papapetrou, *SonarQube in Action*, 1st ed. Greenwich, CT, USA: Manning Publications Co., 2013.
[30] B. Baldassari, "SQuORE: a new approach to software project assessment," Aug. 2013.
[31] L. B. Foganholi, R. E. Garcia, D. M. Eler, R. C. M. Correia, and C. O. Junior, "Supporting Technical Debt Cataloging with TD-Tracker Tool," *Adv Soft Eng*, vol. 2015, pp. 4:4–4:4, Jan. 2015, doi: 10.1155/2015/898514.

[32] C. Fernández-Sánchez, H. Humanes, J. Garbajosa, and J. Díaz, “An Open Tool for Assisting in Technical Debt Management,” in *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Aug. 2017, pp. 400–403, doi: 10.1109/SEAA.2017.60.

[33] T. S. Mendes, F. G. S. Gomes, D. P. Gonçalves, M. G. Mendonça, R. L. Novais, and R. O. Spínola, “VisminerTD: a tool for automatic identification and interactive monitoring of the evolution of technical debt items,” *J. Braz. Comput. Soc.*, vol. 25, no. 1, p. 2, Jan. 2019, doi: 10.1186/s13173-018-0083-1.

[34] “Managing Technical Debt,” *Construx*, Feb. 24, 2017. https://www.construx.com/resources/whitepaper-managing-technical-debt/ (accessed Mar. 30, 2020).

Chapter III. EVOLUTION OF WEB APPLICATIONS

The work of this chapter was published in the Information and Software Technology Journal (IST):

Amanatidis, Theodoros & Chatzigeorgiou, Alexander. (2016). Studying the Evolution of PHP Web Applications. Information and Software Technology. 72. 48-67. 10.1016/j.infsof.2015.11.009.

Chapter Summary

Software evolution analysis can reveal important information concerning maintenance practices. Most of the studies which analyze software evolution focus on desktop applications written in compiled languages, such as Java and C. However, a vast amount of the web content today is powered by web applications written in PHP and thus the evolution of software systems written in such a scripting language deserves a distinct analysis. The aim of the study in this chapter is to analyze the evolution of open-source PHP projects in an attempt to investigate whether Lehman's laws of software evolution are confirmed in practice for web applications. Data (changes and metrics) have been collected for successive versions of 30 PHP projects while statistical tests (primarily trend tests) have been employed to evaluate the validity of each law on the examined web applications. Results suggest that Laws: I (Continuing Change), III (Self-regulation), IV (Conservation of organizational stability), V (Conservation of familiarity) and VI (Continuing growth) are confirmed. However, only for laws I and VI the results are statistically significant. On the other hand, laws II (Increasing complexity), and VIII (Feedback system) do not hold in practice. Finally, for the law that claims that quality declines over time (Law VII) the results are inconclusive. The examined web applications indeed exhibit the property of constant growth as predicted by Lehman's laws and projects are under continuous maintenance. However, no evidence has been found that quality deteriorates over time, a finding which, if confirmed by other studies, could trigger further research into the reasons for which PHP web applications do not suffer from software ageing.

1. Introduction

Scripting languages originated as easy-to-use, specialized, interpreted programming languages supporting loose data typing but quickly evolved to robust, generic and high-level languages boosting the development of the Web [1]. The popularity of scripting languages nowadays is clearly evident from the statistics in open-source repository hosting providers such as SourceForge[7] and GitHub[8]. Languages such as PHP, Javascript, Python, Perl and Ruby are among the most popular choices for developing client and server-side applications, supported by huge communities and vast documentation. PHP in particular has been widely employed in servers around the world as part of the LAMP (Linux-Apache-MySQL-PHP) platform. The top-ten programming languages and the accompanying project share are shown in Table 1 for two open source software repository hosting providers.

The popularity of scripting languages can possibly be attributed to their ease of use, enabling rapid application development and shielding from low-level issues such as memory management [1]. According to Prechelt [2], who contrasted the implementation time for developing in scripting languages (Perl, Python, Rexx and Tcl) with the time for programming the same functionality in

[7] http://sourceforge.net

[8] http://github.com

C/C++/Java, development time for scripting languages is significantly smaller (about half of the time for compiled languages). Scripting languages are being viewed by various authors as more appropriate for real programming pragmatism since they unleash the programmer's creativity and imagination [1]. Back in 1998, Ousterhout [3] claimed that new applications will be written entirely in scripting languages while the so-called system programming languages will be used primarily for developing components.[9]

Table 1. Top-ten Languages of Public Open Source Projects Hosted By SourceForge & Github

SourceForge			**Github**		
Language	# of projects	percentage*	Language	# of repositories	percentage*
Java	53.575	23%	JavaScript	1.666.302	22%
C++	43.189	19%	Java	1.413.447	19%
PHP	33.789	15%	Ruby	888.679	12%
C	31.837	14%	Python	814.449	11%
C#	17.053	7%	PHP	697.898	9%
Python	16.585	7%	CSS	529.392	7%
JavaScript	13.884	6%	C++	439.423	6%
Perl	10.012	4%	HTML	432.546	6%
Unix Shell	4.775	2%	C	386.232	5%
VB .NET	4.050	2%	C#	356.856	5%
Total	228.749	100%	Total	7.625.224	100%

*Percentages refer to the ratio over the total number of projects developed in the top-ten languages

**Data as of October/2015 has been retrieved from http://sourceforge.net and http://github.com

In this chapter the evolution of PHP web applications is investigated, aiming at gaining insight into the way that the corresponding software systems are maintained. The motivations for this dissertation are the following three facts: a) There is a latent perception that scripting languages are not suitable for proper software engineering that can support the maintenance of large-scale software projects [1]. However, such claims can hardly be found in the scientific literature possibly because they are not backed up by real evidence. b) Academics are often skeptical about the suitability of scripting languages in the context of introductory computer science courses. Nevertheless, it should be noted that there is an increasing number of software engineering courses where concepts are illustrated on languages such as Ruby and Python [5]. c) Finally, to the best of the author's knowledge, there is no empirical study investigating the evolution of software projects written in PHP (except for the work in [6]) while there is a large body of research on evolution of software in compiled languages, such as Java.

Software evolution is often studied from the perspective of Lehman's eight laws [7] which characterize trends in size, changes and quality of evolving software systems. Therefore, the main goal of this chapter is to investigate the validity of Lehman's laws of evolution on PHP web applications. Since similar

[9] Nevertheless, the superiority of statically typed languages with respect to maintainability remains open. For example, recent empirical evidence [4] has shown that static types are beneficial to understanding undocumented code and fixing of type errors.

studies have been performed previously for other programming languages, this analysis can be considered as a replication study contrasting previous findings against those derived for PHP.

The rest of the chapter is organized as follows: In Section 2 related work on software evolution and Lehman's laws of software evolution, in particular, is discussed. The details of the case study design are presented in Section 3 along with information about the examined projects. The validity of Lehman's laws of evolution is examined in Section 4. In Section 5 the results are summarized and compared against those of previous works. In Section 6, possible implications for software researchers and practitioners are presented. Threats to validity are discussed in Section 7 and finally, conclusions are drawn in Section 8.

2. Related Work

The analysis of software evolution is one of the most well studied aspects of software development and maintenance. This kind of empirical studies is greatly facilitated by the existence of multiple available data in software repositories allowing the investigation of research questions regarding all facets of a software project, including its source code, documentation, developers, bug reports etc. A comprehensive survey on more than 80 approaches on mining software repositories to investigate aspects of software evolution has been presented by Kagdi et al. [8]. The relation between software evolution and maintenance, highlighting the concept of essential change within an environment, is discussed in the overview paper by Godfrey and German [9].

Software evolution has been studied since the seventies. Lehman first formulated three basic principles of software evolution, based on the study of the OS/360 operating system, in 1974 [10]. Later, Lehman modified the existing principles and proposed two new ones [11]. In the early eighties, Lehman published a new version of Laws III, IV and V [12]. Finally, Lehman published a newer formulation of the laws including additional ones [7] and republished the most current formulations in 2006 [13]. Table 2 lists the most updated formulation of the eight laws of software evolution:

Table 2. Most Updated Formulation of Lehman's Laws

Law	Context
I) Continuing change	*A system must be continually adapted to its users' needs, else it becomes progressively less satisfactory in use.*
II) Increasing complexity	*As a system evolves, its complexity increases and becomes more difficult to evolve unless work is done to maintain or reduce the complexity.*
III) Self regulation	*Global E-type system evolution is feedback regulated.*
IV) Conservation of organizational stability	*The work rate of an organization evolving a software system tends to be constant over time.*
V) Conservation of familiarity	*The newly introduced content of each new version of the system is constrained by the need to maintain familiarity.*
VI) Continuing growth	*The size of a system continuously grows over time.*
VII) Declining quality	*The quality of a system will appear to be declining over time, unless proactive measures are taken.*
VIII) Feedback system	*The evolution process of software resembles a feedback system.*

With the rise of open source software, several studies investigated the validity of the laws and in some cases it was found that some of the laws are not confirmed [14]. Godfrey and Tu, examined the evolution of the Linux Kernel [15] and in later work several other open source systems [16]. Their focus was the

growth of the kernel, using the LOC as size metric and it was found that Linux had been growing at a geometric rate.

Robles et al. [17] examined a wider range of open source systems, including the Linux kernel, as well. In agreement with Godfrey & Tu, they found that smooth growth of systems is not that common and, in some cases, development of open-source software has not followed the laws as known.

In 2008, Mens et al. [18] studied the evolution of Eclipse. They found that laws I and VI were confirmed in practice (i.e. systems are continually adapted at a constant work rate) while law II was not confirmed (i.e. the complexity does not exhibit an increasing trend).

Later, Xie et al. [19] studied the validity of all eight laws of evolution on seven open source projects. They analyzed 653 official releases and cumulatively 69 years of evolution confirming 4 out of 8 laws (I, II, III, VI).

Israeli & Feitelson [20] studied the validation of the laws also on the Linux Kernel in 2010. They found that the superlinear growth found by Godfrey & Tu [15], [16] and confirmed by Robles et al. [17] changed to linear from one point on. Ultimately, they confirmed the *3rd* and *4th* law unlike the aforementioned studies.

In the same year, Businge et al. [21] also examined the validation of the laws on 21 third-party plug-ins of Eclipse. They reached the conclusion that laws I, III and VI are confirmed while V is not.

Later, Neamtiu et al. [22], whose work was an expansion of the study by Xie at al. [19], studied nine open source C projects. The authors validated only the *1st* and the *6th* law, opposing their conclusions in their previous study [19]. In a recent work [23], Kaur et al. studied two C++ projects and found that laws I, II, III, V, VI and VII hold in practice while for IV and VIII they could not reach a safe conclusion.

It is apparent that depending on the examined systems and the approach taken, different laws are confirmed by different studies. A comparative overview of the findings of several studies dealing with the validity of Lehman's laws is provided in Section 5.2 along with the ones observed for PHP code in this chapter.

3. Case Study Design

The objective of the study in this chapter is to examine whether Lehman's laws of software evolution are confirmed in practice for PHP web applications. To achieve this goal data from 30 PHP projects of various sizes and domains have been analyzed. In the following sub-sections, the four parts the design are described. i.e., Goal and Research question, Selection of cases, Employed process and tools and Data analysis.

3.1. Goal and Research Question

The goal of this chapter of dissertation, adopting the formalism of the Goal-Question-Metrics (GQM) approach [24] can be stated as:

Analyze successive versions of web applications written in PHP **for the purpose of** evaluation **with respect to** their evolution **from the perspective of** researchers and software developers *in* **the context of** Lehman's laws of software evolution.

According to this goal the following research question can be formulated, that will guide this dissertation:

RQ: *Is the evolution of web applications written in PHP compliant with Lehman's laws of evolution*?

The research question is then decomposed into eight research questions, one for each of Lehman's laws.

3.2. Selection of Cases

As already mentioned, the chapter focuses on web applications developed with the scripting language PHP. The motivation for selecting web applications was that PHP is primarily used in a Web context and particularly in the widely employed LAMP platform (Linux-Apache-MySQL-PHP). The criteria for selecting the projects are:

- the source code should be publicly available (the code is publicly available if the project is distributed over a source code repository hosting provider, like Github)
- projects should have varying sizes and lifespans to obtain a representative sample (e.g. an almost equal number of projects in three size clusters, 1-10 KLOC, 10-50 KLOC and >50 KLOC has been selected).
- projects should have at least 5 releases in their history to justify evolution analysis (this information is provided by the repositories)
- projects should be object-oriented to allow analysis at the class and method level (this requirement has been checked by counting the number of identified classes using the employed tools)

The projects' source code has been retrieved from Github and Sourcefore because of their large collection of projects and widespread usage. The projects that have been selected for this dissertation are obviously a subset of all projects that satisfy the aforementioned criteria. The large projects in terms of size, namely projects Drupal, Wordpress, laravel, symfony, phpmyadmin and Zendframework, have been selected after discussions with PHP developers who pointed to their importance and indications of high quality. The rest of the projects have been selected by browsing all projects, sorted by relevance and filtering out the ones that did not match the aforementioned criteria. A number of 30 projects has been chosen to enable the manual investigation of the findings and the visual interpretation of the identified trends.

The projects are listed in Table 3 along with an overview of their functionality, their lifespan, size in thousand lines of code and number of analyzed versions. It should be noted that some of the examined projects are relatively small (e.g. Nononsenseforum) while others are large projects with a vast community of developers and users (e.g. WordPress).

Table 3. Overview of Examined Projects

Project	**Functionality**	**Time Frame**	**LOC (last version)**	**Versions**
boardsolution	Discussion board	Jan09 - May13	88k	8
breeze	A micro-framework for PHP 5.3+	Apr13 - Jul13	9k	18
cloudfiles	API for the Cloud Files storage system	Oct09 - May12	5k	13
codesniffer	Code Sniffer tokenizes PHP, JavaScript and CSS files and detects coding standard violations	Nov11 - Sep13	45k	18
conference_ci	EllisLab's Open Source Framework	Aug11 - Oct12	49k	6
copypastedetector	Copy/Paste Detector for PHP code	Jan09 - Aug13	2k	19
dotproject	Web-based project management framework	Aug03 - Nov09	118k	10

Project	Functionality	Time Frame	LOC (last version)	Versions
drupal (core)	Open source CMS	Jan07 - Aug14	18k	61
firesoftboard	Bulletin board software	Mar11 - Nov12	66k	5
generatedata	Random data generator in JS, PHP and MySQL	Jan13 - Sep13	136k	11
laravel	PHP Framework	Feb12 - Mar13	49k	29
mustache	Logic-less template engine	Apr10 - Aug13	7k	33
neevo	Database abstraction layer for PHP 5.3+	Jun11 - Apr13	8k	13
nononsenseforum	Simple discussion forum	Jun11 - Feb13	1k	25
openclinic	Medical records system	Aug04 - Sep13	16k	10
phpagenda	Agenda tool	Sep06 - Jun13	10k	29
phpbeautifier	Parses source code and formats it in preferred styles	Apr05 - Jun10	7k	12
phpdaemon	Asynchronous server-side framework for Web - network applications	Oct10 - Jul13	31k	10
phpfreeradius	Web-based tool for managing a FreeRADIUS environment	Apr10 - Mar12	31k	8
phpmyadmin	Database administration tool	Mar10 - Oct14	252k	68
phpmyfaq	A multilingual, completely database-driven FAQ system	Jan10 - Jul13	88k	49
Phpqrcode	QRCode generator library	Mar10 - Oct10	9k	6
simplephpblog	Blog	Nov05 - Jul12	20k	12
symfony	PHP Framework	Jul11- Oct14	326k	52
tangocms	A modular content management system	Dec09 - Feb12	49k	16
thehostingtool	Client management script geared towards free web hosting providers	May10 - Apr13	27k	6
usebb	Forum system	Feb05 - Jan13	9k	32
web2project	Business-oriented Project Management	Jun10 - Sep13	120k	5
wordpress	Blog tool, publishing platform and CMS	Apr05 - May14	224k	77
zendframework2	PHP Framework	Sep12 - Sep14	284k	25

By definition web applications entail a multitude of technologies. At a first level, web applications contain source code at the server-side (written in PHP in the examined projects) as well as code that takes over the presentation of web pages to clients (written in HTML, CSS, JavaScript etc). Beyond code, a web application contains also other resources (e.g. images, fonts, media files, etc.) accessed by the codebase. It should be mentioned that object-orientation was introduced in version PHP4 and fully supported since version PHP5. However, the typical PHP web application contains both functions as well as classes (methods). To provide an overall picture of this distribution of content types, Figure 1 presents the (a) file and (b) function and method breakdown for the latest release of the examined projects. Approximately half of the files are PHP files and almost 9 out of 10 functions are methods.

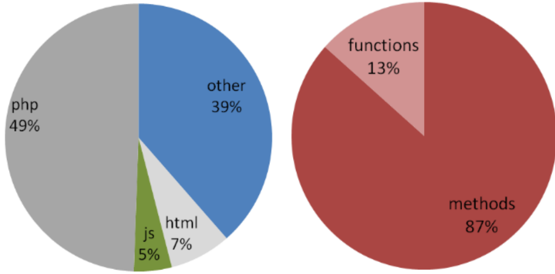


Figure 1. (a) File and (b) function breakdown of examined projects based on their latest release

3.3. Employed Process and Tools

In order to perform the study, a PHP tool has been developed that is capable of parsing the directories of several project releases (uploaded as a single compressed file) and extracting changes between successive releases. Additions, deletions and moves at each level are identified based on the location of the corresponding entity (file, class, function or method), while for the identification of changes the tool examines the percentage of similarity between the body of the same entity in two successive releases (after removing blank lines and comments). The entire workflow is illustrated in Figure 2.

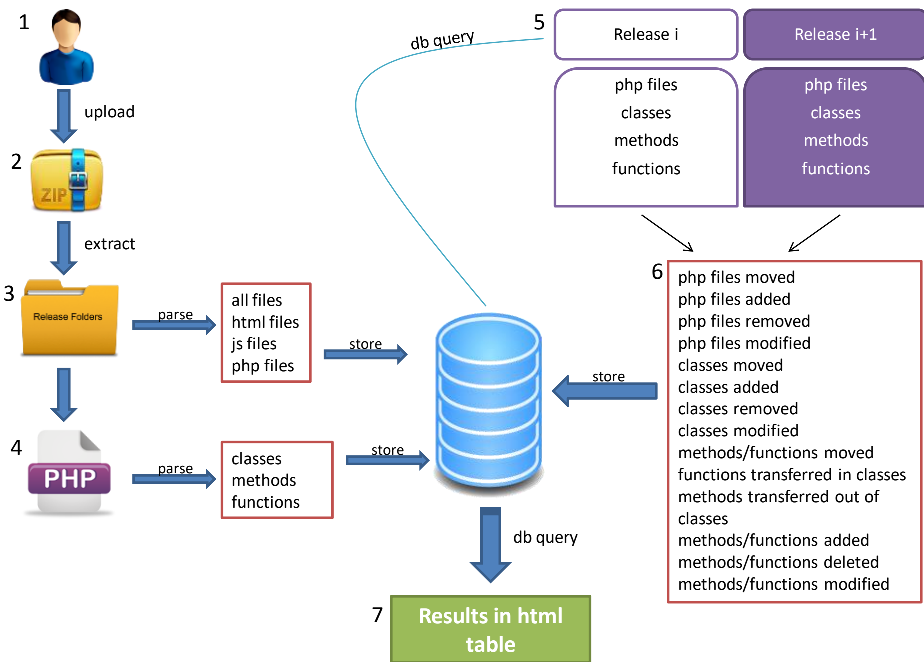


Figure 2. Workflow for analyzing types and frequency of changes in PHP projects

Once information is extracted from the analyzed source code and directory structure (steps 1-4), raw data is stored in a MySQL database. The developed tool also performs the queries to the database considering two successive releases each time (step 5) and changes are stored in the database (step 6). Eventually the tool displays the results in HTML format (step 7).

Moreover, in order to assess the validity of the laws in a quantitative manner, the PHP Depend[10] tool was employed which performs static code analysis and computes several software metrics for PHP applications.

3.4. Data Analysis

As already made clear, the purpose the study in this chapter is to examine whether PHP web applications are evolving in agreement with the Lehman's laws of software evolution. Lehman's laws have been formulated at a rather abstract level, without direct reference (in most cases) to software metrics that can be used to assess them in a quantitative manner [25]. For the mapping of Lehman's laws to measurable indicators the following have been taken into consideration: a) the original formulation or examples provided by Lehman, b) the indicators that have been proposed in previous works that investigated Lehman's laws and c) the suitability of available metrics which can be computed by the employed tool (PHP Depend) for PHP projects. The association between the investigated laws, involved metrics (variables) and the corresponding statistical tests that will be performed to assess the validity of each law is presented in Table 4. Due to plethora of laws, the motivation for the selection of the particular metrics and the analysis conducted for each law will be separately discussed in the Results Section (Section 4).

Table 4. Data Analysis

Laws	**Variables**	**Data analysis**
Law I (Continuing Change)	[V_1] Days Between Releases (DBR)	-Trend test -Slope estimation
Law II (Increasing Complexity)	[V_2] Complexity metric: Cyclomatic Complexity Number / Lines Of Code (CCN/LOC)	-Trend test -Slope estimation
Law III (Self Regulation)	[V_3] Incremental growth of methods & functions	-Trend test -Slope estimation
Law IV (Conservation of organizational stability)	[$V_{4.1}$] Maintenance effort: Effort = total changes / DBR [$V_{4.2}$] Number of commits	-Trend test -Slope estimation
Law V (Conservation of familiarity)	[V_5] Incremental changes (IC) in methods & functions	-Trend test -Slope estimation
Law VI (Continuing growth)	[V_6] Lines of Code (LOC)	-Trend test -Slope estimation

[10] http://pdepend.org/

Law VII (Declining quality)	[$V_{7.1}$] Afferent Coupling (CA)* [$V_{7.2}$] Efferent Coupling (CE)* [$V_{7.3}$] Depth of Inheritance Tree (DIT)* [$V_{7.4}$] Comment Ratio (CR): Commented Lines Of Code / Lines Of Code [$V_{7.5}$] Maintainability Index (MI) [$V_{7.6}$] Number of bug-related commits	-Trend test -Slope estimation
Law VIII (Feedback system)	[V_8] Actual ($\frac{dS}{dt}$) and theoretical growth rate ($c \cdot t^{-\frac{2}{3}}$)	two sample Kolmogorov-Smirnoff test

* These metrics have been measured at class level and their average values (divided by the number of classes) have been considered.

As mentioned above, the study mainly focused on the evolution of these metrics over time. Particularly, the goal was to examine if there is a trend in the evolution of each metric that concerns a specific law and if so, to quantify this trend in comparable numbers. The corresponding null hypothesis for each metric *x* can thus be expressed as:

H_0: Metric x exhibits no trend

H_1: Metric x exhibits a trend

In order to determine if a trend is present in the evolution of a metric the linear regression and the Mann – Kendall trend test [26] were employed. Linear regression is considered a robust modeling tool. However, to consider the results of a trend test based on linear regression as valid, a number of preconditions have to be satisfied. These assumptions are:

1. Variables should be measured at the continuous level (i.e. they should be either interval or ratio variables). Due to the nature of the examined time series of metric values, this condition is always met.
2. The relationship between dependent and independent variables has to be linear.
3. No significant outliers should exist. (The 2nd and 3rd assumption can be assessed visually by examining the scatterplot of the two variables i.e. release number and metric value)
4. Observations should be independent. This can be checked using the Durbin – Watson test which assesses whether residuals of a linear regression model exhibit autocorrelation [27].
5. The data should be characterized by homoscedasticity. This can be checked using the Breusch – Pagan test for homoscedasticity [28].
6. The residuals (errors) of the regression line should be normally distributed. This can be checked by conducting the Shapiro – Wilk test of normality [29] on the residuals of the model yielded from the linear regression.

In case the aforementioned assumptions do not hold, one should use a non-parametric test instead. A trend test which can provide reliable results when no distribution can be assumed is the Mann – Kendall trend test [26].

It should be noted that in the majority of projects one or more assumptions are violated and thus, the Mann – Kendall trend test was mainly used. This is not uncommon when working with real-world data

rather than artificially made examples. When according to the Mann – Kendall trend test a trend is clearly evident, i.e. the null hypothesis can be rejected, the Theil – Sen estimator [30] was used in order to calculate the slope of the fitted trendline. The slope obtained by the Theil – Sen estimator is essentially the median slope among all lines through all pairs of points in the dataset.

To enable the comparison of the steepness of slopes among different projects, slopes should be scale independent. To this end, the trend test analysis (either linear regression or Mann – Kendall trend test) was performed on a normalized version of the original dataset. In particular, each value of an examined time series was divided by the maximum value in the time series yielding a normalized value in the range [0..1] exhibiting the same slope as the original dataset. Moreover, the slope was expressed as a percentage to allow easier interpretation of the results.

Due to the nature of Lehman's laws, many of the variables seem to be akin. Especially the variables related to the 3rd, 4th and 8th law seem to be quite similar. For this reason: a) the difference between variables V_3, $V_{4.1}$, and V_8 was illustrated through a simplified example and b) correlation analysis among all pairs of selected variables for all 30 examined projects was performed.

Figure 3 illustrates a hypothetical system that evolved from version *i* to version *i*+1 over a period of 100 days. For simplicity, it was assumed that 7 new functions (methods and functions) have been added, while 3 existing functions have been modified (removals and moves were counted as changes). The actual values of variables V_3, $V_{4.1}$, and V_8 would then be obtained as shown in the right-hand side of the Figure. As it can be observed these values are indeed closely related but capture different aspects of system evolution.

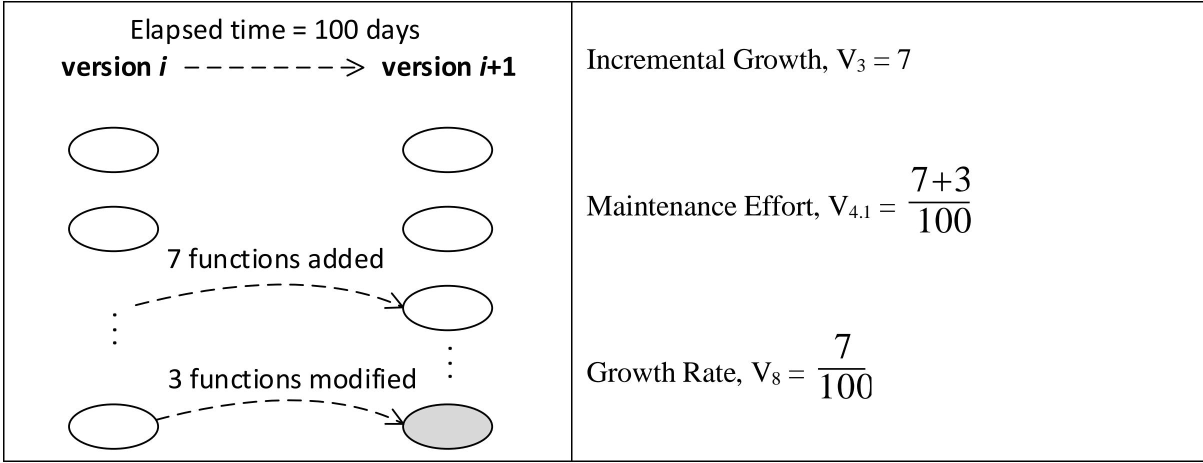


Figure 3. Calculation of incremental growth, maintenance effort and growth rate (example)

To provide further insight into possible correlation between the selected measures, the filled cells in Table 5 indicate cases where the corresponding row and column variables have a statistically significant correlation (with the same sign in the corresponding Pearson's correlation coefficient) in 50% or more of the projects. For example, variable V_2 (CCN/LOC) has a negative correlation to V_6 (LOC) in 19 out of the 30 projects. The average correlation coefficient for these projects is -0.88. This is rather reasonable, since variable V_6 (LOC) is the denominator of variable V_2 (CCN/LOC). However, both variables were deliberately retained, since measuring the complexity of an evolving system would yield a monotonically increasing trend due to the constant addition of new code, as it will be explained in the next section.

Variables $V_{7.1}$ (afferent coupling) and $V_{7.2}$ (efferent coupling) also appear to have a rather strong correlation. However, these variables quantify different aspects of coupling and it was preferred to keep them both in the investigation of the 7th law (nevertheless, it would be worth investigating why these aspects of coupling are correlated in PHP systems).

A strong correlation has been found also between variables $V_{7.2}$ (efferent coupling) and $V_{7.4}$ (comment ratio). This rather unexpected correlation is unexpected, but comment ratio was included in the investigation of quality evolution as it quantifies a distinct property of both functions and methods.

Finally, a strong correlation is observed between the variables discussed in the example of Figure 3, namely between incremental growth (V_3) and growth rate (V_8), and between maintenance effort ($V_{4.1}$) and growth rate. As explained previously, it is reasonable that these variables are correlated as they depend on some common measures. However, because the formulation of the 8th law follows strictly a quantification approach proposed by Turski [31] this variable was not discarded.

Other variables with evidence of a strong correlation to some of the selected ones, have been excluded from the analysis.

Table 5. Correlation Between Variables

	V_1	V_2	V_3	$V_{4.1}$	$V_{4.2}$	V_5	V_6	$V_{7.1}$	$V_{7.2}$	$V_{7.3}$	$V_{7.4}$	$V_{7.5}$	$V_{7.6}$	V_8
V_1														
V_2							19/30 -0.88							
V_3														22/30 +0.769
$V_{4.1}$														18/30 +0.829
$V_{4.2}$														
V_5														
V_6										15/30 +0.858				
$V_{7.1}$									18/30 +0.906					
$V_{7.2}$											18/30 +0.834			
$V_{7.3}$														
$V_{7.4}$														
$V_{7.5}$														
$V_{7.6}$														
V_8														

*Statistical significance is assessed at the 0.05 level

The entire dataset on which the study has been performed is publicly available[11].

[11] http://se.uom.gr/index.php/projects/evolution-analysis-php-applications/

4. Results and Discussion

In this section, the results concerning the research question of whether the evolution of web applications written in PHP is compliant with Lehman's laws of evolution, are going to be presented. To facilitate understanding, a brief reminder of each law will be provided. The hypothesis, the analyzed variables as well as the corresponding type of analysis is also presented for each law. Finally, the rationale behind the selection of the corresponding metrics is explained as well as any concerns that someone could have with the applied approach.

At this point the following clarification should be made: For the laws where the results allow to draw a conclusion that is supported by statistically significant trend test results, the corresponding law is noted as statistically validated or not. However, there are laws, where although the results do not allow the extraction of a statistically significant conclusion, the actual examination of the cases reveals the lack of any evident trend. In these cases, the corresponding law is noted as practically validated or not.

4.1. Law I: Continuing Change

The law states that a program continuously changes and adjusts to its users' needs else it becomes progressively less satisfactory [7]. This is another way of stating that system maintenance is an inevitable process [32]. It is a general observation which is valid for all projects that deliver consecutive releases in a repository, otherwise there wouldn't be a need to release new versions. Law I, is confirmed by all studies on Lehman's laws (see section 5.2 - Comparison with previous work), including this dissertation. The usual way to assess the validity of this law has been to investigate the cumulative number of modified modules [22]. The cumulative number of changed methods and functions in PHP code have also been employed and found a steady increasing trend in all projects, implying that changes are present throughout projects' lifespan. However, a trend is by definition almost always present in a cumulative function, unless no modules are introduced at all during the course of a project, which is rather unlikely. Therefore, the goal was not only to assess the validity of the law *per se*, but also to quantify whether the validity of the law becomes weaker over time or not.

To obtain an insight on whether the first law of Lehman weakens or strengthens over time, the Days Between Releases (DBR) have been measured, denoting the number of days that elapsed from the release of one version in the repository up to the release of the next one. In other words, DBR quantifies the frequency at which new releases are published. An increase of DBR over time means that the rate of publishing new releases decreases, which in turn can be interpreted as a weakening of the validity of the law for a particular project. Thus, the corresponding hypothesis can be expressed as:

<table>
<tr><th>Hypothesis</th><th>Variable</th><th>Analysis</th></tr>
<tr><td>H_0: The evolution of the time interval between two successive releases exhibits no trend.

H_1: The evolution of the time interval between two successive releases exhibits a trend.</td><td>[V_1]: Days Between Releases (DBR)</td><td>-Trend test

-Slope estimation</td></tr>
<tr><td colspan="3">Rationale for selected variable: Previous research has used the cumulative number of modified functions/methods; however, a cumulative number would be monotonically increasing. Therefore, the law is considered valid and the Days Between Releases are selected to assess the frequency at which new releases are published (i.e. whether the law is strengthened over time).</td></tr>
<tr><td colspan="3">Concerns: The elapsed time between releases does not necessarily reflect the amount of changes that have been carried out, especially in open-source projects.</td></tr>
</table>

For example, Figure 4 illustrates the evolution of DBR for the successive versions of project usebb. It appears that the number of days required to release a new version increases over time (less than 50 days for the initial versions which climbs to more than 200 days for the final versions) implying that more effort is required to adapt the system to additional requirements.

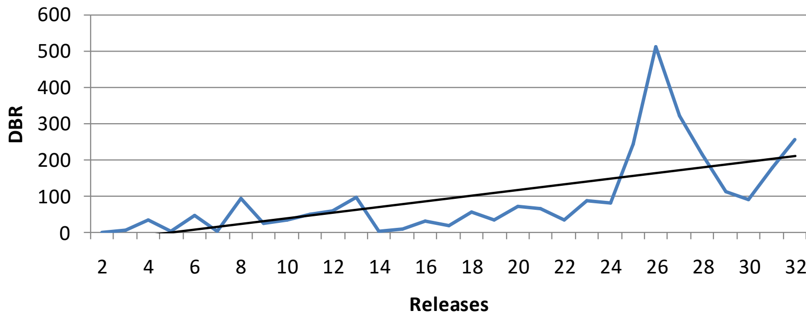


Figure 4. Trend of Days Between Releases metric for project usebb

As already mentioned, to perform a systematic analysis regarding the presence of a trend in a time series, appropriate trend tests and slopes estimation will be used (as explained in section 3.4). Table 6 lists the results of the conducted trend test for each project as well as the slopes for the cases where the trend is statistically significant. In the ‘Trend’ column an up-pointing/down-pointing arrow indicates the presence of a statistically significant trend while a blank cell indicates that there is no evidence for the existence or the absence of a trend.

Table 6. Statistical Results on Law I (Continuing Change)

	Project	**DBR**				**Project**	**DBR**		
		p-value	Trend	Slope (%)			p-value	Trend	Slope (%)
1	boardsolution	0.287			16	phpagenda	0.001	↑	0.07
2	breeze	0.041	↑	0.16	17	phpbeautifier	0.310		
3	cloudfiles	0.086			18	phpdaemon	0.029	↓	-2.78
4	codesniffer	0.366			19	phpfreeradius	0.764		
5	conference_ci	0.462			20	phpmyadmin	0.001	↓	-0.39
6	copypastedetector	0.471			21	phpmyfaq	0.557		
7	dotproject	0.754			22	phpqrcode	0.086		
8	drupal (core)	0.927			23	simplephpblog	0.087		
9	firesoftboard	1.000			24	symfony*	~0.000	↑	0.14
10	generatedata	0.525			25	tangocms	0.546		
11	laravel	0.003	↑	0.83	26	thehostingtool	1.000		
12	mustache	0.025	↑	1.19	27	usebb	~0.000	↑	1.01
13	neevo	0.783			28	web2project	1.000		
14	nononsenseforum	0.274			29	wordpress	0.805		
15	openclinic	0.602			30	zendframework2	0.014	↑	1.25

*Linear regression has been used for these projects and the Mann-Kendall trend test for the rest to obtain p-values.

As it can be observed, in 2 out of the 30 projects DBR decreases over time (i.e. a negative slope is observed) and in 7 out of 30 projects DBR increases. For 21 projects there is no statistical evidence for the existence or the absence of a clear trend. Therefore, one cannot argue about the validity of this law based on statistically significant results. However, to shed light on the evolution of DBR for the majority of the projects that do not exhibit a statistically significant trend, their graphical evolution is depicted in Figure 5. The x-axis corresponds to normalized version numbers, in the sense that all project lifespans are plotted as equal, for the sake of clarity. The y-axis does not contain units, as the curves have been adjusted to minimize their overlap.

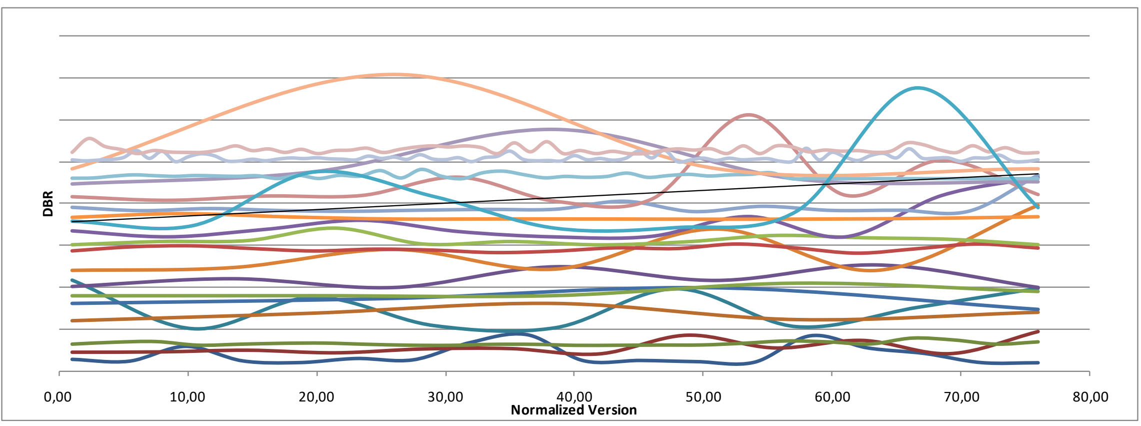


Figure 5. Evolution of Days Between Releases metric for projects with p-value > 0.05

As it can be observed, indeed most of the projects shown in Figure 5 do not exhibit a clear trend but rather have fluctuations in the variable of interest (DBR). One could argue, that DBR does not increase nor decrease steadily during the examined period and characterize this evolution as rather stable.

These observations imply that the first law of Lehman does not become stronger (changes are not becoming more frequent) or weaker over time. In other words, findings suggest that PHP systems continuously change but, in this dissertation, it cannot be determined whether these changes happen at a slower or a faster pace.

4.2. Law II: Increasing Complexity

According to this law the complexity of software increases over time unless proactive measures are taken to reduce or stabilize the complexity [7]. Although the complexity of a software project can be quantified in many ways, the widely acknowledged cyclomatic complexity measure [33] has been selected since it manages to assess the complexity of both functions and methods present in most PHP web applications nowadays. However, the CCN metric provided by the PHP Depend tool counts the total available decision paths in the entire program, and thus would be monotonically increasing as the system becomes larger in size over time. Therefore, its value was normalized over the lines of code, i.e. CCN/LOC. An increase of CCN/LOC over time implies that the overall complexity increases and that the law is valid. The corresponding hypothesis can be expressed as:

Hypothesis	**Variable**	**Analysis**
H_0: The evolution of complexity exhibits no trend. H_1: The evolution of complexity exhibits a trend.	[V_2]: CCN/LOC	-Trend test -Slope estimation
Rationale for selected variable: Cyclomatic complexity is a well-studied and widely acknowledged complexity measure which has also been employed in previous studies for the examination of the validity of the 2nd Law.		
Concerns: The normalization by dividing with the size might not capture changes in total complexity due to the addition of new code.		

The trend of CCN/LOC over all examined versions for each project is shown in Table 7. Figure 6 illustrates the trendline fitted to the evolution of CCN/LOC, for those projects where a statistically significant trend has been found. The x-axis corresponds to normalized version numbers, in the sense that all project lifespans are plotted as equal, for the sake of clarity.

Table 7. Statistical Results on Law II (Increasing Complexity)

	Project	CCN/LOC				Project	CCN/LOC		
		p-value	Trend	Slope (%)			p-value	Trend	Slope (%)
1	Boardsolution	0.319			16	phpagenda	~0.000	↓	-0.40
2	Breeze	0.034	↑	0.03	17	phpbeautifier	0.019	↓	-0.34
3	Cloudfiles	0.853			18	phpdaemon	0.474		
4	Codesniffer	~0.000	↑	1.22	19	phpfreeradius	0.711		
5	conference_ci	0.181			20	phpmyadmin	~0.000	↓	-0.51
6	copypastedetector	0.003	↓	-0.14	21	phpmyfaq	0.016	↓	-0.18
7	Dotproject	0.371			22	phpqrcode	0.035	↓	-0.79
8	drupal (core)	~0.000	↓	-0.86	23	simplephpblog	0.099		
9	firesoftboard*	0.011	↓	-0.02	24	symfony	~0.000	↓	-0.05
10	Generatedata	0.002	↓	-0.17	25	tangocms	~0.000	↑	0.19
11	Laravel	0.763			26	thehostingtool	0.024	↑	2.01
12	Mustache	0.026	↓	-0.60	27	usebb	0.909		
13	Neevo	0.112			28	web2project	0.086		
14	nononsenseforum	~0.000	↑	1.91	29	wordpress	~0.000	↓	-0.20
15	Openclinic	0.149			30	zendframework2	~0.000	↑	0.07

*Linear regression has been used for these projects and the Mann-Kendall trend test for the rest to obtain p-values.

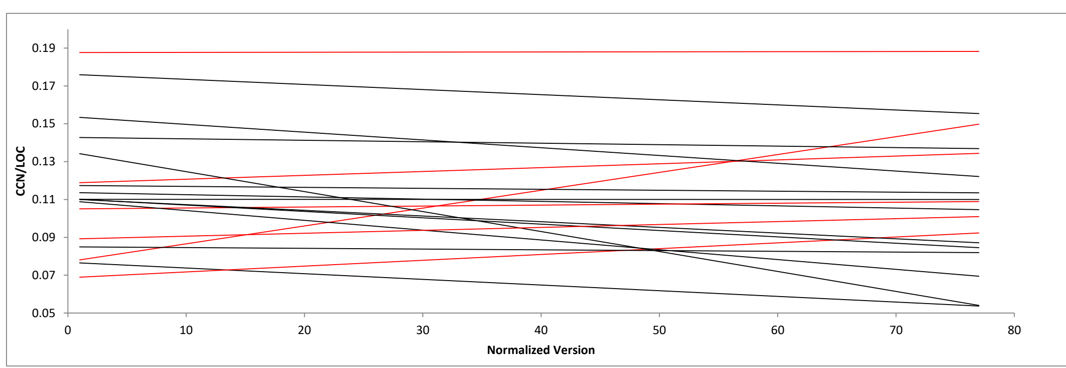


Figure 6. Trendlines of CCN/LOC for projects with p-value < 0.05

As it can be observed from Table 7, in 18 projects (more than half of the projects) there is either a positive or a negative trend in the evolution of the aforementioned complexity measure. Out of the 18 projects in which the null hypothesis is rejected (meaning that a statistically significant trend is present), only in 6 projects there is a deterioration in the evolution of the aforementioned complexity measure, implying that the law is not valid for the examined PHP projects. For the majority of the projects, complexity decreases. This generally decreasing trend is also evident from the CCN/LOC trendlines in Figure 6. To be accurate, it should be reminded that Lehman acknowledged the possibility of a non-increasing complexity if care is exercised by the maintenance team and this seems to be the case for the examined PHP projects. This observation is in agreement with a previous study [6] on the evolution of large-scale PHP web applications, which suggested that systems like phpMyAdmin, WordPress and Drupal exhibit signs of careful maintenance decisions resulting in non-increasing complexity.

4.3. Law III: Self-Regulation

Lehman [7] suggested that "*system evolution process is self-regulating*". In contrast to other rules, mapping this claim to the evolution of quantitative measures is non-trivial. According to Xie et al. [19] the regulation of size throughout the lifespan of a project, translates to observing negative and positive adjustments ("ripples") in the growth trend. The same interpretation of the third law has been adopted by Businge et al. [21] who observed ripples in the incremental growth of Eclipse plugins. To this end, the changes in the total number of functions and methods have been measured. For example, such

changes for project phpMyFAQ are graphically depicted in Figure 7. As it can be observed, ripples are present; positive adjustments are more frequent than negative, in agreement to what has been observed by the study of Xie et al. [19] and Businge et al. [18]. However, no global trend appears to be present. To have a common interpretation of whether the law is confirmed across all projects, whether there is a statistically significant trend in the data was investigated. The law should be considered as invalidated when there is a trend at the incremental growth of the methods and functions of the system. The corresponding hypothesis can be expressed as:

Hypothesis	**Variable**	**Analysis**
H_0: The evolution of incremental growth exhibits no trend. H_1: The evolution of incremental growth exhibits a trend.	[V_3]: incremental growth of methods & functions	-Trend test -Slope estimation
Rationale for selected variable: Methods and functions in PHP code cumulatively reflect the amount of delivered functionality. Incremental growth of system characteristics (e.g. functions, dependencies) has been used in other studies as well.		
Concerns: Evolution might occur at a lower level than methods and functions (i.e. at the code line level) without affecting the number of methods and classes.		

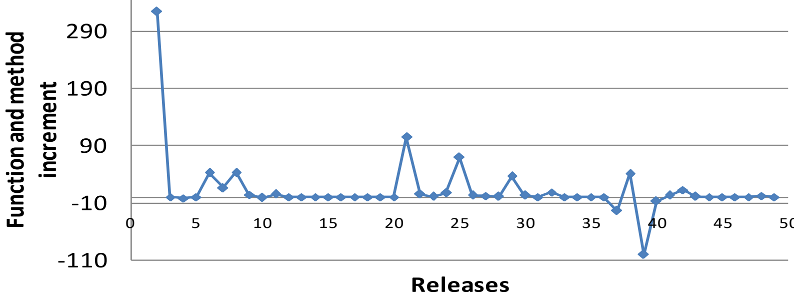


Figure 7. Ripples in the total number of functions/methods for phpMyFAQ

Table 8. Statistical Results on Law III (Self-Regulation)

	Project	INCREMENTAL GROWTH				Project	INCREMENTAL GROWTH		
		p-value	Trend	Slope (%)			p-value	Trend	Slope (%)
1	boardsolution	1.000			16	phpagenda	0.533		
2	breeze	0.426			17	phpbeautifier	0.065		
3	cloudfiles	0.528			18	phpdaemon	0.602		
4	codesniffer	1.000			19	phpfreeradius	0.095		
5	conference_ci	0.579			20	phpmyadmin	0.277		
6	copypastedetector	0.811			21	phpmyfaq	0.285		
7	dotproject	0.016	↓	-0.24	22	phpqrcode	0.267		
8	drupal (core)	0.079			23	simplephpblog	0.436		
9	firesoftboard	0.734			24	symfony	0.011	↑	0.01
10	generatedata	0.653			25	tangocms*	0.118		
11	laravel	0.024	↑	0.04	26	thehostingtool	1.000		
12	mustache	0.960			27	usebb	0.901		
13	neevo*	0.077			28	web2project	0.734		
14	nononsenseforum	0.248			29	wordpress	0.811		
15	openclinic	0.295			30	zendframework2	0.130		

*Linear regression has been used for these projects and the Mann-Kendall trend test for the rest to obtain p-values.

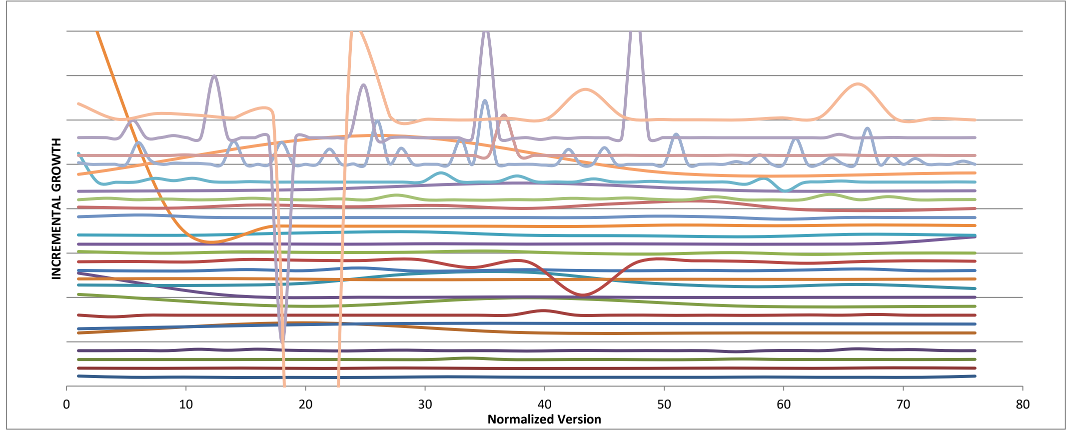


Figure 8. Evolution of Incremental Growth for projects with p-value > 0.05

The results of the statistical analysis are summarized in Table 8. Only in 3 out of the 30 projects a trend in the incremental growth of methods and functions is present. In laravel and symfony there is an increasing trend, meaning that more and more functionality is added over time, while in dotproject the trend is decreasing. In the rest of the projects, the null hypothesis that the incremental growth of the system exhibits no trend cannot be rejected. However, with a closer look at Figure 8, which illustrates graphically the evolution of the incremental growth for all 27 projects where no statistically significant trend has been found, one can observe that indeed there is no evidence for a constant increase or decrease in the number of incremental methods and functions at every new version. This means that the examined systems do grow, but the growth rate remains relatively stable. To sum up, the results are not clear in terms of statistical power that the 3rd Law is valid, but the actual evidence points to the conclusion that the evolution of PHP projects is indeed regulated under a stable growth pace during system's lifespan. Hence, the law is considered as practically validated.

4.4. Law IV: Conservation of Organizational Stability

The law stipulates that the activity/work rate between successive releases remains stable. Estimating effort in open-source projects can hardly be accurate and only indirect measures can be considered. In analogy to the study by Xie et al. [19] the work rate is measured as the number of changes (in the number of methods and functions) in a release i, over the elapsed time (in days) from the previous release i-1. As suggested by Lehman [34], [35] this dissertation counts as changes all handled elements accounting for removed, modified, added and moved functions and methods. Moreover, to provide an alternative measure for the estimation of work rate, the number of commits to the corresponding repository were analyzed, over time. Since a commit implies an 'official' submission of performed work, it can be considered as a reliable indicator of effort. Although this law is considered sub judice (under judgment) in the corresponding study by Lehman, the validity of the law is investigated by assessing the slope of the fitted trendline of maintenance effort, as reflected in the two variables. The statistical results for the trend test on variable V4.1 and V4.2 are shown in Table 9.

Hypothesis	**Variable**	**Analysis**
H_0: The evolution of maintenance effort exhibits no trend. *H_1*: The evolution of maintenance effort exhibits a trend.	[$V_{4.1}$]: maintenance effort = changes/DBR [$V_{4.2}$]: number of commits	-Trend test -Slope estimation

Rationale for selected variables: As suggested by Lehman, the changes in methods and functions throughout a project's lifespan were counted. Moreover, a commit constitutes an actual and 'official' submission of work by the developers.
Concerns: The work that has been performed to release a new version is not reflected accurately when counting source code modifications only, since other types of activities (such as understanding and testing) might have been carried out.

Table 9. Statistical Results on Law IV (Conservation of Organizational Stability)

	Project	**MAINTENANCE EFFORT**			**NUMBER OF COMMITS**		
		p-value	Trend	Slope (%)	p-value	Trend	Slope (%)
1	boardsolution	0.368			0.251*		
2	breeze	0.091			N/A	N/A	N/A
3	cloudfiles	0.732			0.069*		
4	codesniffer	0.711			0.038	↑	0.93
5	conference_ci	0.462			0.007	↓	-1.8
6	copypastedetector	0.622			0.746		
7	dotproject	0.175			0.001	↓	-1.12
8	drupal (core)	0.589			0.189		
9	firesoftboard	0.105*			0.450*		
10	generatedata	1.000			0.463		
11	laravel	0.402			0.002	↓	-2.94
12	mustache	0.023	↓	-0.14	0.194		
13	neevo	0.033	↓	-2.27	0.039	↓	-5.32
14	nononsenseforum	0.049	↑	0.09	0.034	↓	-5.69
15	openclinic	0.754			0.656		
16	phpagenda	0.020	↓	-0.11	N/A	N/A	N/A
17	phpbeautifier	1.000			0.332		
18	phpdaemon	0.016	↑	7.46	0.653		
19	phpfreeradius	0.133			0.033*	↓	-21.16
20	phpmyadmin	0.152			~0.000	↑	0.23
21	phpmyfaq	0.709			0.241		
22	phpqrcode	0.221			N/A	N/A	N/A
23	simplephpblog	0.119			N/A	N/A	N/A
24	symfony	0.033	↑	0.02	0.833		
25	tangocms	0.266			0.634*		
26	thehostingtool	0.807			0.432		
27	usebb	~0.000	↓	-0.86	0.001	↓	-0.83
28	web2project	0.029*	↓	-29.2	0.134		
29	wordpress	1.000			~0.000	↑	1.10
30	zendframework2	0.001	↓	-0.37	0.761		

*Linear regression has been used for these projects and the Mann-Kendall trend test for the rest to obtain p-values.

As it can be observed only in 9 projects (for $V_{4.1}$) and in 10 projects (for $V_{4.2}$) there is a statistically significant trend in the maintenance effort. For the majority of projects, no safe conclusion regarding the evolution of maintenance effort can be reached. Once again, these non-statistically significant cases are depicted in Figure 9 for $V_{4.1}$ and in Figure 10 for $V_{4.2}$.

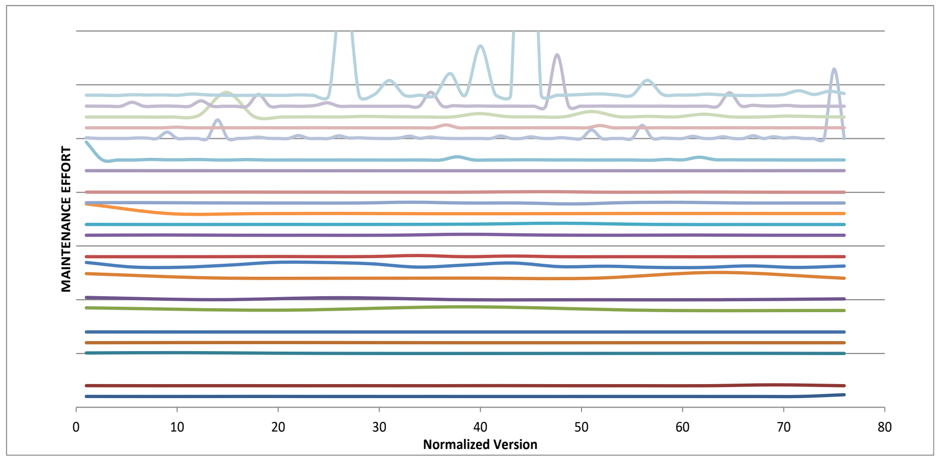


Figure 9. Evolution of Maintenance Effort ($V_{4.1}$) for projects with p-value > 0.05

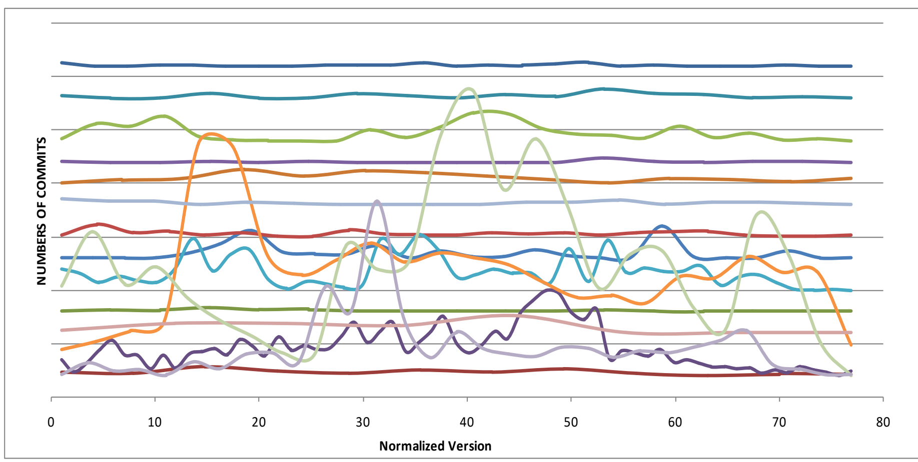


Figure 10. Evolution of Number of Commits ($V_{4.2}$) for projects with p-value > 0.05

The visual interpretation of Figure 9 indicates that in general, the work rate does not increase or decrease drastically as the projects evolve. It should be noted that although some lines appear almost straight, the statistical power was low because of the small number of data points. The evolution of the number of commits in Figure 10 exhibits fluctuations for some of the projects, but again no conspicuous trend is present. Overall, PHP projects seem to evolve in agreement with the 4th Law. An increasing trend would imply that more and more features (or bug fixes) are added to the evolving project in the same period of time, or that the same amount of functionality is added in less and less time. However, it is reasonable to assume that increasing addition of functionality is rather rare for mature open-source projects and especially web applications which have to deliver their core functionality right from their first versions. On the other hand, a decreasing trend would imply that the system suffers from poor maintainability, in the sense that equal amounts of functionality required more time to be added. However, this phenomenon has not been observed meaning that the majority of the examined web applications do not suffer from this kind of maintainability issues. This law is tagged as practically validated.

4.5. Law V: Conservation of Familiarity

According to Lehman, "*During the active life of a program the release content of the successive releases of an evolving program is statistically invariant*" [7]. The law resulted by noticing the inherent tradeoff between the increased difficulty of understanding changes contained in a new release and the organizational pressure for delivering novel features along with the constant demand for corrections

and changes [13]. In order to assess the validity of the law in a quantitative manner, the *Incremental Changes* (IC) metric has been proposed [36]. IC is obtained by subtracting the total number of changes that occurred in methods and functions in one release from the total number of changes in methods and functions of the next release. An absence of trend for IC indicates the absolute validity of the law. A decreasing trend implies that the performed changes become less and less over time, which in turn can be attributed to the increased effort that developers need to understand and modify the program's source code [19].

Hypothesis	**Variable**	**Analysis**
H_0: The evolution of incremental changes exhibits no trend. H_1: The evolution of incremental changes exhibits a trend.	[V_5]: Incremental changes (IC) in methods & functions	-Trend test -Slope estimation
Rationale for selected variable: The incremental changes in methods and functions were measured, as they capture the potential to provide more and more functionality in each new version. If this is not possible, the release content should be considered invariant.		
Concerns: The number of new/modified/deleted functions is only one way of capturing the provision of novel features in a new version.		

Table 10. Statistical Results on Law V (Conservation of Familiarity)

	Project	**INCREMENTAL CHANGES**				**Project**	**INCREMENTAL CHANGES**		
		p-value	Trend	Slope(%)			p-value	Trend	Slope(%)
1	boardsolution	1.000			16	phpagenda	0.872		
2	breeze	0.837			17	phpbeautifier	0.479		
3	cloudfiles	0.627			18	phpdaemon	0.754		
4	codesniffer	0.509			19	phpfreeradius	1.000		
5	conference_ci	1.000			20	phpmyadmin	0.705		
6	copypastedetector*	0.592			21	phpmyfaq	0.986		
7	dotproject	0.917			22	phpqrcode	0.807		
8	drupal (core)	0.753			23	simplephpblog	0.533		
9	firesoftboard	0.308			24	symfony	0.592		
10	generatedata	0.032	↑	0.30	25	tangocms	0.691		
11	laravel	0.634			26	thehostingtool	0.807		
12	mustache	0.770			27	usebb*	0.677		
13	neevo*	0.668			28	web2project	0.734		
14	nononsenseforum	0.823			29	wordpress	0.993		
15	openclinic	0.348			30	zendframework2	0.941		

*Linear regression has been used for these projects and the Mann-Kendall trend test for the rest to obtain p-values.

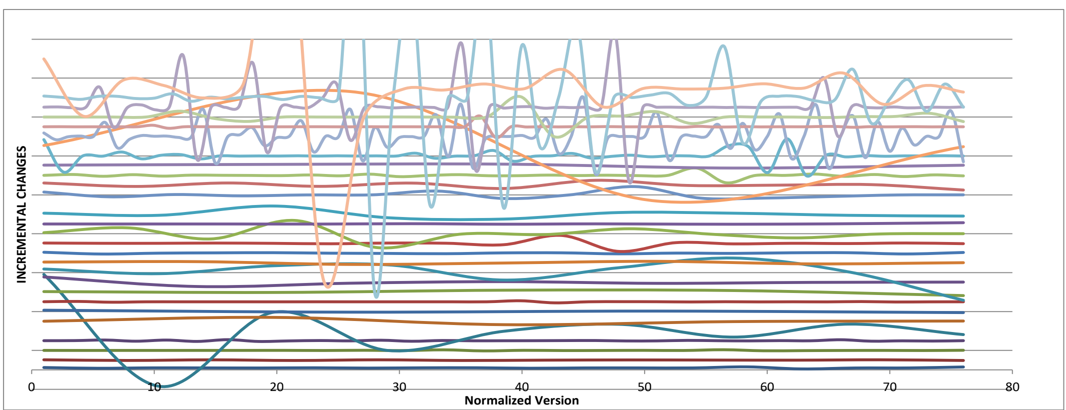


Figure 11. Evolution of Incremental Changes for projects with p-value > 0.05

The results of Table 10 do not allow for reaching a statistically safe conclusion as only in one project a statistically significant trend of IC is evident. For the rest of the projects, trend tests yielded a p-value of more than 0.05 implying that one cannot reject the null hypothesis. For this reason, the actual evolution of these cases was plotted in order to visually check the existence of a trend. As it can be observed in Figure 11, in the majority of the projects, evolution of IC does not exhibit an increasing or a decreasing trend. In other words, the number of additional changes at the method and function level between successive versions might fluctuate temporarily, but is generally invariant over time. This translates to conservation of the release content of each new version in PHP applications which in turn suggests the validity of the 5th law. Thus, this law is tagged as practically validated.

This law is quite similar to the previous one and the findings also match. However, the dimension of time is not taken into account for the 5th law in the sense that the number of incremental changes is not normalized over the elapsed time from the previous release. An increasing trend for the 5th law would imply that the amount of functionality added or modified in each new release is steadily increasing. Such a trend cannot be expected continuously and even if it is present in the initial versions of a new project, it would be unrealistic for mature projects. On the other hand, a decreasing trend would imply that fewer and fewer functions and methods are added or changed over time, signifying a slowly 'dying' project. None of the examined projects exhibits such a trend and it would be worth investigating which kind of actual projects are being gradually abandoned.

4.6. Law VI: Continuing Growth

The law stipulates that a program grows over time to address the new needs of its clients. Although several measures can be employed to assess this growth, most previous studies have used size metrics such as Lines of Code (LOC) [19] or the number of modules [7]. In this dissertation, the evolution of LOC was also measured to capture both additions of statements within functions as well as additions of new functions and classes (methods). An increasing trend for LOC validates the law. The results concerning the trend test are summarized in Table 11, while Figure 12 depicts the corresponding trendlines for the majority of the projects where a statistically significant trend has been found.

Hypothesis	**Variable**	**Analysis**
H_0: The evolution of system's size exhibits no trend. H_1: The evolution of system's size exhibits a trend.	[V_6]: LOC	-Trend test -Slope estimation
Rationale for selected variable: The evolution of the size of each project in terms of LOC was examined, as in most of the previous studies.		
Concerns: -		

Table 11. Statistical Results on Law VI (Continuing Growth)

	Project	LOC				Project	LOC		
		p-value	Trend	Slope (%)			p-value	Trend	Slope (%)
1	boardsolution	0.002	↑	0.19	16	phpagenda	~0.000	↑	0.56
2	breeze*	~0.000	↑	0.91	17	phpbeautifier	~0.000	↑	0.57
3	cloudfiles	0.001	↑	0.91	18	phpdaemon	~0.000	↑	5.86
4	codesniffer	0.256			19	phpfreeradius	0.001	↑	1.98
5	conference_ci	0.566			20	phpmyadmin	~0.000	↑	0.85
6	copypastedetector	~0.000	↑	2.21	21	phpmyfaq	~0.000	↑	0.85
7	dotproject	~0.000	↑	1.59	22	phpqrcode*	0.012	↑	12.90
8	drupal (core)	~0.000	↑	1.63	23	simplephpblog	0.837		
9	firesoftboard	0.807			24	symfony	~0.000	↑	1.08
10	generatedata	~0.000	↑	0.39	25	tangocms	0.051		
11	laravel	~0.000	↑	2.60	26	thehostingtool	0.024	↑	3.40
12	mustache	~0.000	↑	2.86	27	usebb	~0.000	↑	1.87
13	neevo	0.005	↑	1.19	28	web2project	0.807		
14	nononsenseforum	~0.000	↑	2.99	29	wordpress	~0.000	↑	1.27
15	openclinic	0.003	↑	1.76	30	zendframework2	0.293		

*Linear regression has been used for these projects and the Mann-Kendall trend test for the rest to obtain p-values.

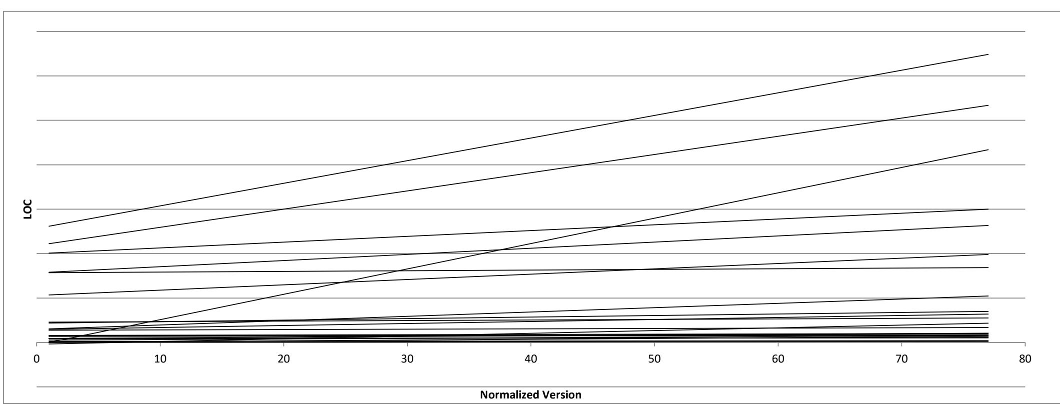


Figure 12. Trendlines of LOC for projects with p-value < 0.05

From the results of Table 11 and the trendlines in Figure 12, it becomes apparent that in the majority of PHP projects (23/30), the size in terms of LOC increases steadily over time. Although deletions of code also occur, in the examined web applications it is evident that development teams keep adding new code to enhance the offered functionality. As a result, one can reach the conclusion that the *6th* law of software evolution holds in practice. This law has been confirmed in all previous studies (see section 5.2 - Comparison with previous work).

4.7. Law VII: Declining Quality

The law states that the quality of software deteriorates over time unless proactive measures are taken. Degradation of software quality over time is a widely investigated phenomenon known under different names, such as "software ageing" [37] or accumulation of technical debt [38]. A number of internal quality metrics and one external quality indicator have been examined to evaluate the validity of this law for PHP applications. Specifically, metrics which can be calculated at the level of individual classes were investigated and which can be associated to an aspect of design quality. Moreover, two metrics were also included. These metrics concern both functions and methods to assess the quality of non-object-oriented code as well. Finally, the number of bug related commits was measured to assess whether the number of bugs increases or decreases over time. In order to avoid any misleading statistical interpretations, only a trend test on the evolution of each metric was performed without attempting to

extract an overall statistical measure considering all metrics. A brief discussion of the employed metrics follows next.

Coupling is one of the classic internal metrics used to assess the quality of a design and for this reason the average Afferent Coupling (CA) and the average Efferent Coupling (CE) of each class were measured. Afferent coupling refers to the number of unique incoming dependencies for a software artifact (i.e. it is representative of a class' fan-in). Therefore, it is an indicator of the extent by which a module is used by other modules, and under normal circumstances, it is suggested to keep the fan-in high [39]. Typical examples of modules/packages with high fan-in are core packages and components, like error and exception handling, or unit testing framework classes.

Efferent coupling counts the number of software artifacts that a software entity depends on. A high efferent coupling (i.e. the module has a high fan-out) implies that the component depends on several other implementation details and this makes the component itself instable, because an incompatible change between two versions or a switch to a different library may break the dependent component. Moreover, the comprehensibility and reusability of a module with high efferent coupling is limited. Therefore it is considered a good practice to keep the efferent coupling for all artifacts at a minimum [39].

The quality of an object-oriented design has also been assessed from the perspective of inheritance qualities. Although specific thresholds for the optimum depth of an hierarchy are hard to extract by means of empirical studies, Harrison and Counsell [40] have found that deeper inheritance trees are harder to understand and maintain, a view shared also in the early discussions on inheritance heuristics by Riel [41]. In this dissertation, the evolution of the 'Depth of Inheritance Tree' metric (DIT) was tracked as PHP systems evolve.

Several studies assess the understandability of code (which is a sub-characteristic of maintainability) by the comment ratio (CR) that is the ratio of commented lines of code over the total lines of code. The higher the ratio for a piece of code is, the more readable and thus maintainable the code can be considered to be [42]. This metric allows to assess the evolution of both function and methods and has been selected as the fourth internal quality indicator.

Another widely used and discussed measure of quality is the Maintainability Index (MI) which has been originally introduced by Oman and Hagemeister in 1991[43]. MI is a composite metric that considers for an assessed module its Halstead's volume, cyclomatic complexity and size in terms of lines of code. There have been numerous studies on the validity of MI, some of which have found that MI can successfully predict actual maintenance effort and others which have questioned its accuracy. Nevertheless, in this dissertation MI was used as an indicator of internal quality because it is not restricted to object-oriented code, and because that regardless of its accuracy as a maintainability predictor, an increasing trend of MI would imply efforts to improve three aspects of quality within functions or methods.

Finally, since all the aforementioned metrics focus on internal quality, a measure that aims at addressing quality as perceived by users or developers was also included. An indisputable indicator of external quality would be the number of bugs/errors found during system evolution, as an increasing number of bugs implies quality degradation. However, although the examined applications are supported by an issue tracking system, for the examined PHP projects, it was found that it would be unreliable to count the number of issues (since in numerous cases the reported issues do not concern bugs). For this reason, the commits (i.e. actual code changes) for which it could be inferred that they are related to the fixing of a bug or issue were selected. As in other studies (e.g. [44]) bug related commits were identified by filtering those that contain error related keywords, such as 'error', 'bug', 'fix' and 'issue' in the corresponding commit message.

For measures CA, CR and MI an increasing trend implies that quality is improving from this perspective. On the other hand, for measures CE, DIT and number of bug related commits, quality is improving if their values get lower. In Table 12, the trend of the aforementioned quality measures over all examined versions for each project is reported. To facilitate the interpretation of the results, the cases in which the evolution of a metric suggests deterioration of the system's quality were marked with shaded cells.

Hypothesis	**Variable**	**Analysis**
H_0: The evolution of system's quality exhibits no trend. H_1: The evolution of system's quality exhibits a trend.	[$V_{7.1}$]: CA [$V_{7.2}$]: CE [$V_{7.3}$]: DIT [$V_{7.4}$] CR [$V_{7.5}$] Maintainability Index (MI) [$V_{7.6}$] Number of bug-related commits	-Trend test -Slope estimation
Rationale for selected variables: The assessment of quality evolution is based on a mixture of internal quality metrics (for object-oriented and procedural code) and one external quality indicator related to the number of bugs. The selected metrics have been tested for correlation among them, as explained in Section 3.4		
Concerns: Internal quality metrics do not necessarily map to external quality. The number of bug-related fixes is sensitive on the style of commit messages employed in a project.		

Table 12. Statistical Results on Law VII (Declining Quality)

	Project	CA			CE			DIT		
		p-value	*Trend*	*Slope(%)*	*p-value*	*Trend*	*Slope(%)*	*p-value*	*Trend*	*Slope(%)*
1	boardsolution	0.003	↑	0.08	0.022	↑	0.07	0.067		
2	breeze	0.128			0.969			~0.000	↑	0.35
3	cloudfiles	0.260			0.260			0.014	↑	0.05
4	codesniffer	0.096			0.185			~0.000	↓	-6.29
5	conference_ci	0.105			0.411			0.105		
6	copypastedetector	0.885			0.017	↑	0.59	~0.000	↑	1.60
7	dotproject	0.105			0.358			0.006	↑	1.66
8	drupal (core)	1.000			0.207			~0.000	↑	0.96
9	firesoftboard	0.613			0.129			1.000		
10	generatedata	0.012	↓	-0.12	0.024	↓	-0.14	0.012	↑	0.25
11	laravel	~0.000	↓	-0.67	0.008	↓	-0.43	~0.000	↑	2.00
12	mustache	~0.000	↑	2.62	~0.000	↑	2.49	~0.000	↓	-1.02
13	neevo	1.000			0.009	↑	1.43	0.001	↑	0.34
14	nononsenseforum	~0.000	↑	5.55	~0.000	↑	5.00	~0.000	↓	-3.94
15	openclinic	0.021	↑	2.98	0.001	↑	7.14	0.165		
16	phpagenda	~0.000	↓	-1.21	~0.000	↓	-1.08	~0.000	↓	-0.90
17	phpbeautifier	~0.000	↑	1.49	0.823			0.148		
18	phpdaemon	0.088			0.059			0.009	↑	4.98
19	phpfreeradius	0.421			0.789			0.421		
20	phpmyadmin	0.004	↑	0.08	0.475			~0.000	↑	0.81

	Project	CA			CE			DIT		
		p-value	*Trend*	*Slope(%)*	*p-value*	*Trend*	*Slope(%)*	*p-value*	*Trend*	*Slope(%)*
21	phpmyfaq	0.359			0.045	↓	-0.13	0.005	↓	-0.16
22	phpqrcode	1.000			0.008	↑	6.91	1.000		
23	simplephpblog	0.453			0.015	↑	14.00	0.078		
24	symfony	~0.000	↑	0.10	~0.000	↑	0.07	~0.000	↑	0.13
25	tangocms	0.021	↓	-0.05	0.006	↑	0.04	0.498		
26	thehostingtool	~0.000	↑	3.81	0.181			0.100		
27	Usebb	1.000			1.000			1.000		
28	web2project	0.267			0.267			0.149		
29	Wordpress	~0.000	↑	0.68	~0.000	↑	0.52	~0.000	↑	1.27
30	zendframework2	~0.000	↑	0.21	~0.000	↑	0.40	~0.000	↓	-0.18

Table 12. (continued) Statistical Results on Law VII (Declining Quality)

	Project	CR			MI			BUG COMMITS		
		p-value	*Trend*	*Slope(%)*	*p-value*	*Trend*	*Slope(%)*	*p-value*	*Trend*	*Slope(%)*
1	boardsolution	0.018	↑	0.01	~0.000*	↑	1.76	0.529		
2	breeze	~0.000	↓	-0.08	~0.000	↓	-0.95	N/A	N/A	N/A
3	cloudfiles	0.358			~0.000	↓	-0.15	0.064*		
4	codesniffer	0.019	↓	-0.07	0.502			~0.000	↑	1.38
5	conference_ci	0.848			~0.000*	↑	0.12	0.691		
6	copypastedetector	0.888			0.772			0.117		
7	Dotproject	0.032	↓	-0.22	N/A	N/A	N/A	0.678		
8	drupal (core)	~0.000	↑	1.17	0.186			~0.000	↑	0.38
9	firesoftboard	0.807			~0.000*	↑	1.30	0.945		
10	generatedata	0.008	↓	-0.28	0.024	↑	1.06	0.002*	↓	-11.8
11	laravel	~0.000	↓	-0.6	0.044	↑	0.09	0.008	↓	-3.34
12	mustache	0.466			0.025	↑	1.07	0.591		
13	neevo	0.005	↓	-0.44	~0.000*	↑	1.9	0.212*		
14	nononsenseforum	~0.000	↑	1.12	~0.000	↑	3.81	1.000		
15	openclinic	0.243			~0.000*	↓	-0.16	N/A	N/A	N/A
16	phpagenda	~0.000	↑	0.32	1.000			N/A	N/A	N/A
17	phpbeautifier	0.002	↓	-0.2	0.115			0.066		
18	phpdaemon	0.127			0.001	↑	5.98	0.212		
19	phpfreeradius	0.004	↓	-0.21	0.035	↓	-3.48	N/A	N/A	N/A
20	phpmyadmin	0.013	↓	-0.06	0.026	↑	0.03	~0.000	↑	0.86
21	phpmyfaq	~0.000	↓	-0.5	~0.000	↓	-0.15	0.446		
22	phpqrcode	0.085			0.011*	↓	-18.9	N/A	N/A	N/A
23	simplephpblog	0.002	↑	2.44	~0.000*	↑	10.01	N/A	N/A	N/A
24	symfony	0.003	↑	0.02	~0.000	↑	0.12	~0.000	↑	4.18
25	tangocms	~0.000	↓	-0.08	~0.000*	↑	0.15	0.837*		
26	thehostingtool	~0.000*	↑	1.53	~0.000*	↓	-6.38	0.065		
27	usebb	0.009	↑	0.36	0.022	↓	-0.23	0.003	↓	-1.17
28	web2project	~0.000*	↑	1.66	0.051*			0.155		
29	wordpress	~0.000	↑	0.84	~0.000	↑	1.54	~0.000	↑	0.83
30	zendframework2	0.441			0.003	↑	0.12	0.112		

*Linear regression has been used for these projects and the Mann-Kendall trend test for the rest to obtain p-values.

As it can be observed from the number of projects in which a statistically significant trend has been found, the overall picture is rather mixed across the examined quality indicators. For afferent coupling quality is increasing in 10 out of the 14 projects and for the maintainability index quality is increasing in 15 out of the 23 projects with a statistically significant trend. Quality is decreasing in 12 out of 16 projects for efferent coupling and in 12 out of 18 projects for the depth of inheritance. In terms of comment ratio in about half of the 21 projects quality is increasing and for the rest quality decreases. For bug related commits, a trend was found only in 8 out of the 20 projects.

The picture is mixed even if table is analyzed horizontally that is, by examining each project separately to identify how often the quality of a project deteriorates or improves over time. Thus, there is no supporting evidence neither for the confirmation nor for the confutation of the 7th law. In other words, it cannot be claimed in general that the quality of the examined PHP projects is declining or improving over time.

4.8. Law VIII: Feedback System

The corresponding claim was stated in 1980 but has been formalized as a law in 1996 [7]. According to Lehman [34], the evolution process of software resembles a feedback system. In other words, the size of a software system in a given release can be described in terms of the size in the previous release and the effort for developing the new release. Turski [31] formulated a model suggesting that the growth of a system, in terms of number of changed modules, is sub-linear, slowing down during the evolution of the project, exactly because the system becomes larger and more complex. The number of modules is preferred over low-level measures such as LOC since according to Turski system functionality changes are reflected in added, removed or otherwise handled modules, a view shared by Lehman in his early studies [13]. Turski proposed a difference equation according to which the size of version *i* can be estimated as:

$$S_i = S_{i-1} + \frac{\overline{E}}{S_{i-1}^2} \quad (1)$$

where (interpretation is fitted to the case of PHP applications):

S_i is the size of version i measured in number of methods and functions and,

$\overline{E}$ is the effort spent on the development of each software release, which is considered constant according to the fourth law of Lehman.

The intuition behind this formulation is that the larger the size of a version, the greater the resistance to change it, in analogy to the effect of mass in a mechanical system or capacity in an electrical system.

Later, Turski generalized the model to a differential form [45] and extracted a closed form for the growth equation as:

$$S(t) = a \cdot t^{\frac{1}{3}} + b \quad (2)$$

where α and b are constants.

By obtaining the derivative of the growth equation, the corresponding rate of growth is:

$$\frac{dS}{dt} = c \cdot t^{-\frac{2}{3}} \quad (3)$$

where:

c is a constant,

and t is the elapsed time (in days) from the initial release.

If the law holds in practice, the rate of growth should be proportional to $t^{-\frac{2}{3}}$, so it is relatively straightforward to check its validity. The actual evolution of $\Delta S/\Delta t$ for all successive release pairs, can be compared to the theoretical evolution by employing the two-sample Kolmogorov-Smirnoff test [46].

As an example, let us consider the evolution of the growth rate for project *mustache* (Figure 13). The solid line represents the observed changes in the growth rate ($\Delta S/\Delta t$), while the dashed line corresponds to the evolution predicted by Lehman's *8th* law according to Turski's model. As it becomes evident the actual $\Delta S/\Delta t$ trend line is well above the rate predicted by the law and the growth rate is not declining as predicted. For this case one can conclude (by visual examination) that the law is not confirmed for this particular project.

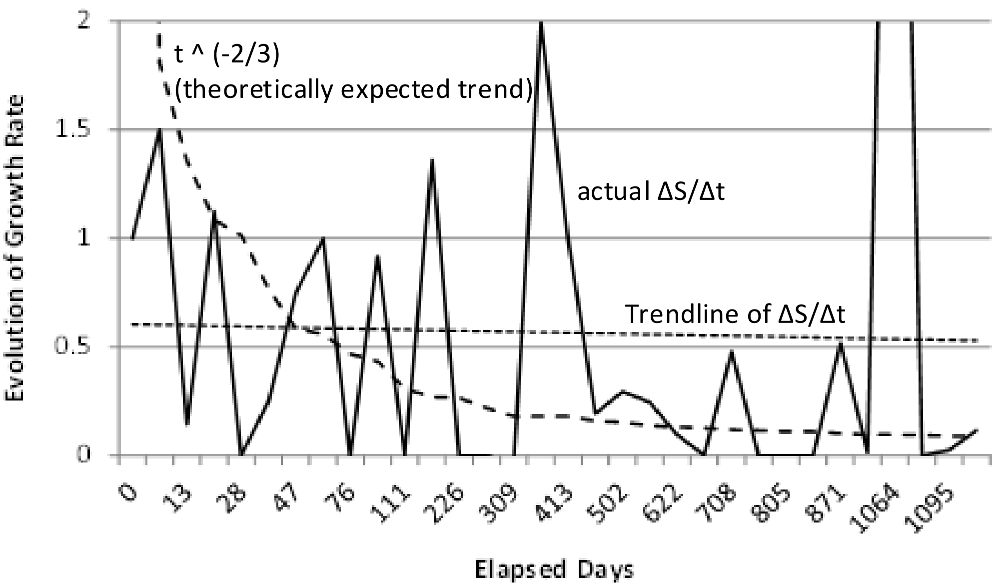


Figure 13. Examination of the validity of the *8th* law in project "*mustache*"

Hypothesis	Variable	Analysis
H_0: The empirically observed rate of growth matches the theoretically expected one. H_1: The empirically observed rate of growth does not match the theoretically expected one.	[V_8]: rate of growth	two sample Kolmogorov-Smirnoff test
Rationale for selected variable: The evolution of the rate of growth of each project was examined and compared against the theoretical one, as proposed by Turski and shared by Lehman.		
Concerns: The primary concern here is the interpretation of the notion of feedback system. In this dissertation the mathematical interpretation provided by Turski [31] is adopted.		

The results from the statistical investigation of the validity of the 8th law are presented in Table 13, listing the significance value of the Kolmogorov – Smirnoff test conducted for each project in order to examine whether the actual growth rate ($\Delta S/\Delta t$) matches the theoretically expected rate. A significance value less than 0.05, means that the null hypothesis can be rejected, implying that the law is not confirmed (the corresponding cases are shaded in the Table).

Table 13. Statistical Results on Law VIII (feedback system)

	Project	Kolmogorov – Smirnoff p-value		Project	Kolmogorov – Smirnoff p-value
1	boardsolution	0.541	16	phpagenda	0.000
2	breeze	0.006	17	phpbeautifier	0.206
3	cloudfiles	0.249	18	phpdaemon	0.000
4	codesniffer	0.000	19	phpfreeradius	0.203
5	conference_ci	0.082	20	phpmyadmin	0.000
6	copypastedetector	0.001	21	phpmyfaq	0.000
7	dotproject	0.002	22	phpqrcode	0.329
8	drupal (core)	0.000	23	simplephpblog	0.023
9	firesoftboard	0.699	24	symfony	0.000
10	generatedata	0.001	25	tangocms	0.003
11	laravel	0.007	26	thehostingtool	0.819
12	mustache	0.002	27	usebb	0.000
13	neevo	0.100	28	web2project	0.211
14	nononsenseforum	0.000	29	wordpress	0.000
15	openclinic	0.699	30	zendframework2	0.000

The growth rate does not match the theoretical expectation in 19 out of 30 projects as marked by the shaded rows in Table 13. Thus, one could argue that the law is not confirmed by the results for the examined PHP applications. In other words, the rate of increase in project size indeed attenuates over time, however, not at the fast rate predicted by Turski's model. It should be noted that the outcome for this law is not in contrast to the findings for the 5th law and 6th law. The results for Law V suggested that one cannot claim that more and more (or less and less) code (incremental changes) is practically added in successive versions, without however considering the time elapsed between releases, whereas the results for Law VI confirmed that systems continuously grow. The findings for this law, which assumes that software processes operate as a feedback system where current size dictates the rate of increase in the next release, suggest that the growth rate is attenuating, i.e. that if time is taken into account, less code is added in a given amount of time. In other words, as the examined applications mature either there is less left to be added in terms of functionality or the system size prevents the development team from keeping the same pace of adding new code. Nevertheless, system development is slowing down at a rather low rate.

5. Overview and Comparison to Previous Work

5.1. Summary of Results

To facilitate the interpretation of the findings regarding the eight laws of Lehman, the corresponding claims are summarized in Table 14 and compared against the results for the examined PHP applications. The laws are grouped in three categories based on the generic aspect/property that they address. As it can observed, from the two laws (II & VII) concerning the evolution of quality the 2nd has not been confirmed for the examined PHP applications while for the 7th law the results were inconclusive. With respect to the laws discussing changes in an evolving system (I, IV & V) the results suggest that all laws are confirmed (the 1st with statistical significance while the other two only at a practical level). In other words, systems continuously undergo changes but no trend has been observed for the work rate or the incremental changes. As a general observation one could claim that the examined PHP applications are maintained without reaching any maintenance stagnation.

Finally, with respect to the laws that address the growth of an evolving system (III, VI & VIII), systems indeed continuously grow and exhibit positive and negative adjustments of incremental growth. However, it could not be confirmed that the growth rate decreases according to the theoretically prescribed rate. In other words, the examined PHP applications do get bigger, are maintained and there

are no clear signs of quality degradation or improvement. Further research into the reasons that drive this evolution patterns of PHP web applications would be extremely valuable.

Table 14. Summary of findings about Lehman's laws

Property	Law	Lehman claims:	Finding (PHP)
Quality	II	*Complexity increases*	Complexity does not increase
	VII	*Quality declines*	Inconclusive results
Changes	I	*System continuously change*	Indeed
	IV	*Work rate remains stable*	Indeed (no statistical significance)
	V	*Incremental changes remain invariant*	Indeed (no statistical significance)
Growth	III	*Incremental growth exhibits negative and positive adjustments (systems are self-regulated)*	Indeed (no statistical significance)
	VI	*Systems continuously grow*	Indeed
	VIII (Turski's form)	*Growth rate decreases at a rate proportional to* $t^{-2/3}$	Growth rate does not decrease that fast

The present study has not been designed to identify the reasons for which certain laws are confirmed for some projects while others are violated. Nevertheless, it is reasonable to assume that the reason for which PHP web applications continuously change and grow is to provide novel services and features to clients in the shortest time possible. This is a necessity in order to withstand the competition caused by the perpetuous outspread of the Web. Such a competitive environment is normally driving the accumulation of the so-called 'Technical Debt' [47]. In other words, speeding-up development time normally compromises software quality, thereby hindering its sustainability. However, this accumulation of Technical Debt is not evident for PHP web applications which manage to evolve without increasing their complexity and without demanding increased effort. This phenomenon could be attributed to the productivity of the language, which allows developers to rapidly produce functional code, and to the widespread usage of reliable libraries and frameworks.

5.2. Comparison to Previous Work

An overview of the approach and the findings regarding the validity of the eight laws of Lehman in previous research is provided in Table 15 and Table 16, along with the results in this dissertation. To provide insight into the approach that has been employed by each research group for the quantification of the examined laws, Table 15 briefly outlines the corresponding measures used in 8 previous studies. (When a law is not investigated in the context of a work, the corresponding cell is left blank). Because of the way that the laws have been stated, as it can be observed from Table 15, the employed measures vary. However, there are laws which are quantified by most of the studies in the same or in a similar manner. For example, law VI is quantified by most of the studies using the LOC metric, and Law III is quantified mainly through the number of functions. On the other hand, law VII, which does not specify which aspect of quality has to be considered, is quantified through a variety of quality indicators.

Table 15. Primary Measures Employed for the Investigation of Laws in Previous Studies

Ref.	I	II	III	IV	V	VI	VII	VIII
Godfrey & Tu*	SLOC			SLOC		SLOC		SLOC
Robles et al.	SLOC			SLOC		SLOC		SLOC
Mens et al.	File Changes	LOC, additions/ modifications, #defects, CC				Several size measures including LOC		
Xie et al.	Cumulative #changes, type of changes	CC, function calls, Coupling	# functions	changes per day, handled functions/total functions	#modules, new functions	LOC, #functions, #definitions	#defects, defect density, complexity measures	#functions
Israeli & Feitelson	#source files	CC	#files	percentage of handled files	Releases per month, intervals between releases	#system calls, #configuration options	Maintainability Index	No quantitative approach
Businge et al.	Cumulative number of added/deleted dependencies		#dependencies		Percentage of handled files, percentage of added dependencies	unique dependencies	Indicator of balance between abstractness and stability	
Neamtiu et al.	cumulative changes	calls per function CC coupling	#modules #functions	changes per day change rate growth rate	net module growth #new functions #changes	LOC #modules #definitions	#defects defect density calls per function CC coupling	#modules LOC #functions
Kaur et al.	#functions and #classes	CBO, RFC, WMC, DIT, LOCH	#functions and #classes	No quantitative approach	#functions and #classes	LOC, #functions and #classes	CC	No quantitative approach
This dissertation	Days between releases	CC	#functions	Maintenance effort and #commits	#functions	LOC	CA, CE, DIT, CR, MI, bug-related commits	#functions

* CC: cyclomatic complexity

* SLOC: source lines of code (uncommented lines of code)

* CBO: coupling between objects

* RFC: Response for class - #methods being invoked in response to the message received by an object of that class

*WMC: weighted methods per class - the sum of the complexities of its methods

* DIT: depth of inheritance tree
* LOCH: lack of cohesion
* CA: coupling afferent (#unique incoming dependencies for a software artifact)
* CE: coupling efferent (#unique outgoing dependencies for a software artifact)
* CR: comment ratio
* MI: Maintainability Index

To allow a comparison with the conclusions derived in other studies about Lehman's laws (which however have not focused on PHP web applications), Table 16 lists the findings from the aforementioned 8 previous studies. A '✓' symbol indicates confirmation, a '×' symbol indicates that the law has not been validated, while the '~' symbol implies that the results have been inconclusive. When a law is not investigated in the context of a work, the corresponding cell is left blank. It should be noted that Table 16 lists the conclusions as derived by the authors of the corresponding papers (for the studies by Godfrey & Tu [15] and by Robles et al. [17] the validity of the 1st , 6th and 8th law is not directly investigated but can be easily deduced from the provided information).

As it can be observed, the 1st law regarding continuing change and the related 6th law on continuing growth are, as expected, validated by all studies. In some studies system growth rate (in LOC) is found to be exponential [15] while in others linear [17]. In other words, all studies agree that systems continuously change and grow (a phenomenon called 'perpetual development' in the study by Israeli and Feitelson [20]). An agreement is also observed between previous studies and the current one for the 2nd and the 8th law. Concerning increasing complexity, in 3 out of the 5 previous works that examined this law and reached conclusive results, it had not been confirmed, as in the case of PHP projects. Concerning the decline of growth rate at the pace predicted by the 8th law, four previous studies (out of the five that reached conclusive results for C/C++/Java projects) found that the actual growth rate attenuates at a slower pace, as it has also been found in this dissertation for PHP projects.

Table 16. Validity of Lehman's Laws According to Various Studies

Ref.	Year	Prog.Lang.	#Projects	I	II	III	IV	V	VI	VII	VIII
Godfrey & Tu*	2000	C	1	✓			×		✓		×
Robles et al.	2005	C,C++, Java	19	✓			×		✓		×
Mens et al.	2008	Java	1	✓	×				✓		
Xie et al.	2009	C	7	✓	✓	✓	~	×	✓	×	×
Israeli & Feitelson	2010	C	1	✓	×	✓	✓	~	✓	×	✓
Businge et al.	2010	Java	21	✓		✓		×	✓	~	
Neamtiu et al.	2013	C	9	✓	×	×	×	×	✓	×	×
Kaur et al.	2014	C++	2	✓	✓	✓	~	✓	✓	✓	~
This dissertation	2015	PHP	30	✓	×	✓**	✓**	✓**	✓	~	×

*The results in a later work by Godrfrey & Tu [16] confirmed the validity of the same laws on 4 projects.

**These laws have not been statistically validated. The conclusion in these cases is based on a visual interpretation of the evolution for the projects where the null hypothesis (absence of trend) could not be rejected.

6. Implications for Researchers and Practitioners

Although the research question that has been set, regarding the validity of Lehman's laws of evolution for PHP web applications, entails a theoretical perspective and thus the results are not directly exploitable, the following implications can be identified.

With respect to software practitioners and managers:

- In the context of the investigation of Lehman's laws of evolution the employed measures can be used to assess the evolution of other products and examine whether any striking deviations from Lehman's observations are valid for their projects. Since most laws are not directly quantifiable, software maintainers could employ the same methodology with respect to the applied trend tests and indicators that have been analyzed for each law.
- Especially with respect to the evolution of quality vs. the increase of size contrasting the results for their own projects to those of the examined applications could highlight issues that warrant attention. For example, it should be regarded as a warning if their own PHP web projects do not success in allowing continuous changes combined with a non-increasing complexity, since this trend has been observed both for small and large open-source projects in this dissertation. If, for example, a development team observes that complexity is constantly increasing, whereas large and complicated PHP systems manage to keep complexity stable or even reduce it over time, then, quality assurance should focus on ways to address the increasing complexity.
- The results suggesting that PHP web applications conform to a lifecycle model where continuous and steady development takes places (a finding confirmed by other studies as well), imply that development teams should opt for agile development practices, where constant change is embraced, rather than models assuming elaborate and preconceived specifications and planning [20].
- The results indicating that PHP web applications continuously change and grow, a finding shared by all other studies as well, imply that project managers should anticipate increased future needs for resources to maintain and sustain the existing systems.

With respect to software engineering researchers:

- Based on the findings indicating that PHP web applications do not suffer from software ageing, researchers can focus on the reasons that drive this improved behavior of PHP projects and investigate whether this is due to the language, the domain or the practices in web application development.
- Researchers are encouraged to investigate whether the same trends are valid for the evolution of systems written in other scripting languages so as to investigate whether similar maintenance patterns can be attributed to the nature of the employed languages (i.e. scripting vs. compiled).
- Finally, for the specific group of research efforts that investigate the validity of Lehman's laws, empirical findings that suggest that: a) several laws are consistently not confirmed (e.g. Law VIII), or that b) some laws occasionally lead to inconclusive results (e.g. Laws IV and VII) or that c) some laws are quantified by divergent approaches (e.g. Law IV), imply that the rules might need to be examined in the context of contemporary software development and possibly be revisited.

7. Threats to Validity

The investigation of the validity of Lehman's laws is by definition threatened by the subjectivity in the interpretation of each law and the selection of appropriate metrics to quantify its evolution. The fact that the employed measures might not reflect accurately the phenomenon under investigation poses a threat to the relation between theory and observation, i.e. to construct validity [48]. In addition, for several laws there might be additional measures that can be used to quantify the corresponding evolutionary trend, which are either not available (such as the effort spent in an open-source project) or unreliable if collected automatically (such as the number of issues). For example, law VII on the evolution of quality, can be quantified by numerous internal and external quality indicators, as it becomes evident from the multitude of metrics employed by previous studies shown in Table 15. To mitigate this threat, for most of the laws this dissertation relied on measures that have been used in previous studies as well. Moreover, to emphasize this inherent limitation in the quantification approach the relevant concerns along with the approach for each law were explicitly stated.

The conclusions derived from any empirical study that is based on a set of examined software systems are subject to external validity threats. In our case, this threat limits the possibility to generalize the

findings regarding the validity of Lehman's laws in PHP applications beyond the 30 examined projects and to other programming languages. In other words, it is not granted that the selected projects are representative of the entire PHP web application landscape. As it is always the case, further replication studies would be extremely valuable. The emphasis on PHP was placed on purpose, since the goal of this chapter was to investigate patterns of evolution in web applications built upon a scripting language. To this regard, further studies could extend the analysis to other primarily scripting languages such as Python, Perl and Ruby.

Finally, since the presented empirical study relies heavily on the interpretation of statistical test results (mainly trend tests) threats to statistical conclusion validity may arise. The conclusions about the identified trends are based on the number of projects that exhibited statistically significant trends. For example, in the 2nd law the normalized complexity was considered to exhibit a trend because a decreasing trend has been observed in 12 out of the 18 projects with a statistically significant result. Such a finding might imply low statistical power. In other words, although the trend test for each project is correctly applied by analyzing the relevant assumptions, one has to aggregate the findings for all projects to reason about the validity of the law. To facilitate the interpretation of the results all data which have led to the confirmation of confutation of each law have been provided.

8. Conclusions

The evolution of software projects relying on scripting languages such as PHP has received limited attention, despite the fact that PHP forms the basis upon which a huge number of web applications are developed. Driven by the widely spread but undocumented claims that scripting languages are not suitable for regularly maintained software projects, an empirical study on the evolution of 30 PHP web applications has been performed in this chapter.

The main goal was to examine the validity of the eight laws of software evolution as stated by M. M. Lehman. These laws have been extensively studied in the context of software evolution for projects developed in compiled languages such as C and C++ and in a non-web related context. The results confirm the validity of continuing growth and changes for the evolution of the examined PHP applications. However, for the examined projects the 2nd law on increasing complexity and the 8th law on the rapid decrease of the growth rate have not been confirmed. Although the root causes for this trend require further investigation it is reasonable to assume that this phenomenon could be attributed either to the programming language or to the practices in web application development.

One interesting line of further research would be to compare the evolution of web applications against that of "conventional" desktop systems, in order to investigate whether there are differences in the trends of quality, work rate, complexity and size. Such evidence would be helpful in determining whether development practices for web applications adhere to the principles of building large-scale, multi-person, multi-version software systems or whether the benefits is the result of their architecture, which is often strictly dictated by the platforms being used.

References

[1] R. P. Loui, "In Praise of Scripting: Real Programming Pragmatism," *Computer*, vol. 41, no. 7, pp. 22–26, Jul. 2008.

[2] L. Prechelt, "Are Scripting Languages Any Good? A Validation of Perl, Python, Rexx, and Tcl against C, C++, and Java," in *Advances in Computers*, vol. Volume 57, Elsevier, 2003, pp. 205–270.

[3] J. K. Ousterhout, "Scripting: higher level programming for the 21st Century," *Computer*, vol. 31, no. 3, pp. 23–30, Mar. 1998.

[4] S. Hanenberg, S. Kleinschmager, R. Robbes, É. Tanter, and A. Stefik, "An empirical study on the impact of static typing on software maintainability," *Empir. Softw. Eng.*, vol. 19, no. 5, pp. 1335–1382, Dec. 2013.

[5] "Python is Now the Most Popular Introductory Teaching Language at Top U.S. Universities." [Online]. Available: http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-us-universities/fulltext. [Accessed: 09-May-2015].

[6] P. Kyriakakis and A. Chatzigeorgiou, "Maintenance Patterns of Large-Scale PHP Web Applications," in *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2014, pp. 381–390.

[7] M. M. Lehman, "Laws of software evolution revisited," in *Software Process Technology*, C. Montangero, Ed. Springer Berlin Heidelberg, 1996, pp. 108–124.

[8] H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *J. Softw. Maint. Evol. Res. Pract.*, vol. 19, no. 2, pp. 77–131, Mar. 2007.

[9] M. W. Godfrey and D. M. German, "The past, present, and future of software evolution," in *Frontiers of Software Maintenance, 2008. FoSM 2008.*, 2008, pp. 129–138.

[10] M. M. Lehman, *Programs, cities, students: Limits to growth?* Imperial College of Science and Technology, University of London, 1974.

[11] M. Lehman, "Laws of Program Evolution-Rules and Tools for Programming Management," in *Proceedings Infotech State of the Art Conference, Why Software Projects Fail?*, 1978, pp. 11/1–11/25.

[12] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proc. IEEE*, vol. 68, no. 9, pp. 1060–1076, Sep. 1980.

[13] N. H. Madhavji, J. Fernandez-Ramil, and D. Perry, *Software Evolution and Feedback: Theory and Practice*. John Wiley & Sons, 2006.

[14] I. Herraiz, D. Rodriguez, G. Robles, and J. M. Gonzalez-Barahona, "The Evolution of the Laws of Software Evolution: A Discussion Based on a Systematic Literature Review," *ACM Comput Surv*, vol. 46, no. 2, pp. 28:1–28:28, Dec. 2013.

[15] M. W. Godfrey and Q. Tu, "Evolution in Open Source Software: A Case Study," in *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, Washington, DC, USA, 2000, p. 131–.

[16] M. Godfrey and Q. Tu, "Growth, Evolution, and Structural Change in Open Source Software," in *Proceedings of the 4th International Workshop on Principles of Software Evolution*, New York, NY, USA, 2001, pp. 103–106.

[17] G. Robles, J. J. Amor, J. M. Gonzalez-Barahona, and I. Herraiz, "Evolution and growth in large libre software projects," in *Eighth International Workshop on Principles of Software Evolution*, 2005, pp. 165–174.

[18] T. Mens, J. Fernandez-Ramil, and S. Degrandsart, "The evolution of Eclipse," in *IEEE International Conference on Software Maintenance, 2008. ICSM 2008*, 2008, pp. 386–395.

[19] G. Xie, J. Chen, and I. Neamtiu, "Towards a better understanding of software evolution: An empirical study on open source software," in *IEEE International Conference on Software Maintenance, 2009. ICSM 2009*, 2009, pp. 51–60.

[20] A. Israeli and D. G. Feitelson, "The Linux kernel as a case study in software evolution," *J. Syst. Softw.*, vol. 83, no. 3, pp. 485–501, Mar. 2010.

[21] J. Businge, A. Serebrenik, and M. van den Brand, "An Empirical Study of the Evolution of Eclipse Third-party Plug-ins," in *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, New York, NY, USA, 2010, pp. 63–72.

[22] I. Neamtiu, G. Xie, and J. Chen, "Towards a better understanding of software evolution: an empirical study on open-source software," *J. Softw. Evol. Process*, vol. 25, no. 3, pp. 193–218, Mar. 2013.

[23] T. Kaur, N. Ratti, and P. Kaur, "Applicability of Lehman Laws on Open Source Evolution: A Case study," *Int. J. Comput. Appl.*, vol. 93, no. 18, pp. 40–46, May 2014.

[24] V. R. Basili, “Software Modeling and Measurement: The Goal/Question/Metric Paradigm,” University of Maryland at College Park, College Park, MD, USA, 1992.
[25] J. Fernandez-Ramil, A. Lozano, M. Wermelinger, and A. Capiluppi, “Empirical Studies of Open Source Evolution,” in *Software Evolution*, Springer Berlin Heidelberg, 2008, pp. 263–288.
[26] H. B. Mann, “Nonparametric Tests Against Trend,” *Econometrica*, vol. 13, no. 3, pp. 245–259, Jul. 1945.
[27] J. Durbin and G. S. Watson, “Testing for Serial Correlation in Least Squares Regression: I,” *Biometrika*, vol. 37, no. 3/4, pp. 409–428, Dec. 1950.
[28] T. S. Breusch and A. R. Pagan, “A Simple Test for Heteroscedasticity and Random Coefficient Variation,” *Econometrica*, vol. 47, no. 5, pp. 1287–1294, Sep. 1979.
[29] S. S. Shapiro and M. B. Wilk, “An Analysis of Variance Test for Normality (Complete Samples),” *Biometrika*, vol. 52, no. 3/4, pp. 591–611, Dec. 1965.
[30] H. Theil, “A Rank-Invariant Method of Linear and Polynomial Regression Analysis,” in *Henri Theil’s Contributions to Economics and Econometrics*, B. Raj and J. Koerts, Eds. Springer Netherlands, 1992, pp. 345–381.
[31] W. M. Turski, “Reference Model for Smooth Growth of Software Systems,” *IEEE Trans Softw Eng*, vol. 22, no. 8, pp. 599–600, Aug. 1996.
[32] I. Sommerville, *Software Engineering*, 9 edition. Boston: Addison-Wesley, 2010.
[33] T. J. McCabe, “A Complexity Measure,” *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.
[34] M. M. Lehman, “Software’s future: managing evolution,” *IEEE Softw.*, vol. 15, no. 1, pp. 40–44, Jan. 1998.
[35] M. M. Lehman, D. E. Perry, and J. F. Ramil, “On evidence supporting the FEAST hypothesis and the laws of software evolution,” in *Software Metrics Symposium, 1998. Metrics 1998. Proceedings. Fifth International*, 1998, pp. 84–88.
[36] S. Ali and O. Maqbool, “Monitoring software evolution using multiple types of changes,” in *International Conference on Emerging Technologies, 2009. ICET 2009*, 2009, pp. 410–415.
[37] D. L. Parnas, “Software Aging,” in *Proceedings of the 16th International Conference on Software Engineering*, Los Alamitos, CA, USA, 1994, pp. 279–287.
[38] Z. Li, P. Avgeriou, and P. Liang, “A systematic mapping study on technical debt and its management,” *J. Syst. Softw.*, vol. 101, pp. 193–220, Mar. 2015.
[39] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*, 2 edition. Upper Saddle River, N.J: Prentice Hall, 2002.
[40] R. Harrison, S. Counsell, and R. Nithi, “Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems,” *J. Syst. Softw.*, vol. 52, no. 2–3, pp. 173–179, Jun. 2000.
[41] A. J. Riel, *Object-Oriented Design Heuristics*, 1 edition. Reading, Mass: Addison-Wesley Professional, 1996.
[42] K. K. Aggarwal, Y. Singh, and J. K. Chhabra, “An integrated measure of software maintainability,” in *Reliability and Maintainability Symposium, 2002. Proceedings. Annual*, 2002, pp. 235–241.
[43] P. Oman and J. Hagemeister, “Metrics for assessing a software system’s maintainability,” in *, Conference on Software Maintenance, 1992. Proceerdings*, 1992, pp. 337–344.
[44] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, “A Large Scale Study of Programming Languages and Code Quality in Github,” in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, New York, NY, USA, 2014, pp. 155–165.
[45] W. M. Turski, “The Reference Model for Smooth Growth of Software Systems Revisited,” *IEEE Trans Softw Eng*, vol. 28, no. 8, pp. 814–815, Aug. 2002.
[46] D. J. Sheskin and D. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures, Second Edition*, 2 edition. Boca Raton: Chapman and Hall/CRC, 2000.
[47] P. Kruchten, R. L. Nord, and I. Ozkaya, “Technical Debt: From Metaphor to Theory and Practice,” *IEEE Softw.*, vol. 29, no. 6, pp. 18–21, Nov. 2012.
[48] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Springer Science & Business Media, 2012.

Chapter IV. TECHNICAL DEBT AND CORRECTIVE MAINTENANCE

The work of this chapter was published in the Information and Software Technology Journal (IST):

Amanatidis, Theodoros & Chatzigeorgiou, Alexander & Ampatzoglou, Apostolos. (2017). The Relation between Technical Debt and Corrective Maintenance in PHP Web Applications. Information and Software Technology. 87. 10.1016/j.infsof.2017.05.004.

Chapter Summary

Technical Debt Management (TDM) refers to activities that are performed to prevent the accumulation of Technical Debt (TD) in software. The state-of-research on TDM lacks empirical evidence on the relationship between the amount of TD in a software module and the interest that it accumulates. Considering the fact that in the last years, a large portion of software applications are deployed in the web, the study of this chapter focuses on PHP applications. Although the relation between debt amount and interest is well-defined in traditional economics (i.e., interest is proportional to the amount of debt), this relation has not yet been explored in the context of TD. To this end, the aim of this chapter is to investigate the relation between the amount of TD and the interest that has to be paid during corrective maintenance. To explore this relation, a case study on 10 open source PHP projects was performed. The obtained data have been analyzed to assess the relation between the amount of TD and two aspects of interest: (a) corrective maintenance (i.e., bug fixing) frequency, which translates to *interest probability* and (b) corrective maintenance effort which is related to *interest amount*. Both interest probability and interest amount are positively related with the amount of TD accumulated in a specific module. Moreover, the amount of TD is able to discriminate modules that are in need of heavy corrective maintenance. The results of the study confirm the cornerstone of TD research, which suggests that modules with a higher level of incurred TD, are costlier in maintenance activities. In particular, such modules prove to be more defect-prone and consequently require more (corrective) maintenance effort.

1. Introduction

In recent years, Technical Debt Management (TDM) has become a popular research field in software engineering. The majority of TDM approaches are based on the two pillars of Technical Debt (TD) quantification, namely ***principal*** (i.e., the effort needed to refactor the system in order to address existing inefficiencies) and ***interest*** (i.e., the additional effort needed in performing maintenance, due to the existence of the principal). According to Alves et al. [1], interest can be perceived as a risk for software development, and therefore its quantification should be assessed based on two components: ***interest probability*** (i.e., how possible is that one module that holds TD will need maintenance) and ***interest amount*** (i.e., the amount of additional effort). According to Ampatzoglou et al. [2] interest is incurred while performing two types of maintenance activities: (a) bug-fixing (namely ***corrective maintenance***), and (b) adding new features (namely ***perfective maintenance***).

In the literature, one can identify several studies that have investigated the relation between low levels of design-time qualities (e.g., coupling, bad smells, etc.) that constitute proxies of modules' ***TD amount***—i.e., principal plus interest—and the maintenance intensity on these modules [3]–[9]:

- All studies agree that the more flaws a file is involved in, the higher the likelihood to undergo defect-related changes.

- MacCormack and Sturtevant have found evidence on 2 industrial projects that source files with higher levels of coupling are associated with more extensive corrective maintenance [3].
- Feng et al. [4] and Nord et al. [5] have found evidence that files participating in architectural flaws (especially in unstable interfaces) are highly correlated with bugs and changes.
- In an earlier study in 2013 [6], Zazworka et al. suggested that dispersed coupling, god class symptoms, modularity violations and multithread correctness issues are located in classes with higher defect-proneness.
- Another work by Li et al. [7] suggests that two modularity metrics are strongly correlated with commit density: IPCI (Index of Package Changing Impact) & IPGF (Index of Package Goal Focus). Strong correlation was also found between corrective maintenance and fan-out, file size and frequency of changes of file in a study by Schwanke et al. in 2013 [8].
- An interesting study has been carried out by Oliva et al. [9], who searched for symptoms of increased rigidity and fragility on a degraded software system (Apache Maven 1.x) which was completely rewritten to Maven 2.x. The authors found signs of increased fragility (i.e. tendency of a system to break when changes are performed), but no definite evidence of increased rigidity (i.e. difficulty in performing changes due to ripple effects).

The results of these studies, despite the fact that some of them are only indirectly related to TD, have produced some evidence about the relation between maintenance effort and TD. However, the following limitations have been identified:

- Almost all studies quantify TD by means of few metrics, whereas TD manifests itself through a number of parameters in a software project.
- Most studies conducted research on a restricted sample of projects limiting the generalizability of the results (except for [4] and [7] that considered 10 and 13 projects, respectively).
- There is no relevant study that focuses on PHP web applications, which form the majority of operating code in Web today.
- There is no study that focuses on the interest that incurs when performing corrective maintenance.

Based on the abovementioned limitations, the purpose of the study in this chapter is to provide insights into the relation between the accumulated amount of TD in a module and the maintenance effort spent on corrective activities. In particular, the relation between TD amount and: (a) frequency of corrective maintenance activities (***interest probability***), and (b) the effort spent in these activities (***related to interest amount***), is investigated. To overcome the limitations mentioned in the previous paragraph: (a) TD amount is calculated with SonarQube[12] that assesses TD based on a seven axes of code quality (e.g., code duplications, metrics, styling conventions, etc.), (b) the case study is performed on 10 open source PHP web applications, and (c) both interest probability and interest amount are holistically investigated.

2. Case Study Design

In this section, the case study design is presented, which based on the guidelines reported by Runeson et al. [10].

2.1. Goal and Research Questions

The goal of this chapter of dissertation is to examine whether the frequency and the effort spent on corrective maintenance activities of a specific module, is related to the amount of its TD. Based on this goal, the main research question of this chapter can be formulated as follows: "*Is the amount of TD in a software module related to the frequency and extent of corrective maintenance activities performed*

[12] Available at: http://www.sonarqube.org

in it?" To ease the reporting of the case study, from this main question, two research questions were derived:

RQ_1: *Is the TD amount of a file related to the number of times that it underwent corrective maintenance?*

RQ_1 aims at investigating whether files with higher amount of TD are associated with more problems and therefore require more frequent corrective maintenance. The presence of such an association would imply that TD can serve as an indicator for prioritizing maintenance and testing activities. Moreover, such a finding would validate the importance of TD as a crucial parameter to be taken into account during software development.

RQ_2: *Is the TD amount of a file related to the extent of modification that it underwent during corrective maintenance?*

Although the number of times a file undergoes corrective maintenance is a solid indicator of interest probability, to investigate whether modules with high TD produce more interest, in RQ_2, the focus placed on the extent of maintenance effort, as captured by the number of modified lines.

It should be clarified that the proposed case study design does not support the investigation of causal relationships between the TD incurred in one revision and the amount of corrective maintenance in subsequent revisions. Such an analysis is an interesting research topic but should be properly performed, since it would be extremely difficult to associate changes in a specific commit, to the TD as measured in one out of the many past revisions.

2.2. Cases and Units of Analysis

This dissertation focuses on web applications developed with PHP. The motivation for focusing on PHP is that it holds the lion's share of operating Web applications today. The criteria for selecting the projects are:

- the source code should be publicly available (data was retrieved via GitHub's API)
- projects should be actively maintained (until the date on which this paper is written)
- projects should have at least 10 releases denoting jumps in functionality or the addition of significant fixes13 in their history to justify evolution analysis
- projects should be popular (among the projects with most stars in GitHub)

The list of the investigated projects (i.e., cases) is presented in Table 1. This part dissertation is an embedded multiple-case study, because it analyzes every project at the file level (unit of analysis), whereas the results are presented at the case (i.e., project) level. The rationale to use files as a unit of analysis was based on the fact that both object-oriented and non-object-oriented code is included. Thus, the use of any other type of module (e.g., class) would not be possible. An alternative to this dissertation design could be to perform a per-version analysis, i.e. by considering the TD amount of each file for every project version and the corrective maintenance between successive versions, as a unit of analysis. However, in such a case the TD of each version would be correlated to the TD of previous versions, thus rendering the data points not independent.

[13] The rationale for this choice is that any project with a history spanning more than 10 significant releases underwent substantial adaptive maintenance and is highly probable to have been the subject of corrective maintenance as well.

Table 1. Analyzed Projects

Project	#stars	#releases
CodeIgniter	12K	27
Symfony	12K	209
Composer	8K	24
Yii2	8K	13
Guzzle	7K	108
Slim	7K	74
Laravel (kernel)	6K	192
Piwik	6K	429
PHPunit	5K	402
Twig	3K	86

2.3. Data Collection

For each unit of analysis (file), three variables were recorded. To facilitate the following description of variables related concepts are illustrated in Figure 1:

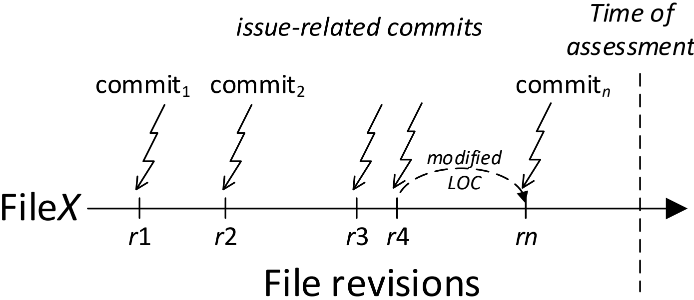


Figure 1. Corrective maintenance at file level

[V1] Average TD of each file: All projects were analyzed with SonarQube and TD of each file (in minutes) was retrieved via the SonarQube. SonarQube calculates a file's TD by summing up the Technical Debt of every violation found on that file, which is the estimated time to fix that violation. The debt for each file represents various aspects of TD quantification, ranging from programing convention violations (e.g., lack of comments) to structural characteristics of the software (e.g., method complexity), which can affect the maintainability and comprehensibility of files. Since several revisions of each file have been analyzed, the TD for each file is obtained as the average of the TD corresponding to the file after each issue-related commit:

$$TD_{FileX} = \frac{1}{n} \cdot \sum_{i=1}^{n} TD(r_i) \tag{1}$$

The average for a file's TD offers the advantage of obtaining a relatively accurate estimate, compared to alternatives, as it characterizes the entire history of the file. On the contrary, assuming that TD remains relatively stable and considering only the TD of the initial or the last revision would not be accurate since by nature software systems are evolving and TD changes over time.

[V2] Number of times each file is modified due to corrective maintenance: Variable V2 corresponds to the total number of issue-related commits (for the examined file) from the initial revision to the time of assessment.

[V3] Number of modifications (modified LOC) each file undergoes during corrective maintenance: Variable V3 corresponds to the average number of modified LOC (for the examined file) from the initial to the last issue-related commit prior to the time of assessment.

To calculate [V2] and [V3] commit and issue data for each project was retrieved via the GitHub API. For each project's issue the commit by which the issue was closed was tracked and eventually found the files that were modified and the number of modified lines in that file. GitHub identifies issue-related commits by recognizing in the commit message the keywords 'fixes', 'resolves' and 'closes' when accompanied by a hash-tagged issue id. The tool for analyzing GitHub data is available online[14].

2.4. Data Analysis

To answer the research questions stated in Section 2.1, using the data described in Section 2.3, correlation analysis and hypothesis testing were performed. For both questions, the same analysis was performed, but on different variables. For RQ_1 the testing variable is [V2], whereas for RQ_2 the testing variable is [V3]. An over-view of the data analysis strategy is presented in Table 2.

Table 2. Data Analysis

RQ	Analysis Strategy
RQ_1	Spearman Correlation [V1] and [V2] Mann-Whitney U Test for [V2] grouped by [V1]
RQ_2	Spearman Correlation [V1] and [V3] Mann-Whitney U Test for [V3] grouped by [V1]

Additionally, the Mann-Whitney U Test (the independent sample t-test was not used, since variables do not follow the normal distribution) is able to investigate the discriminative power of the TD amount as an indicator of corrective maintenance frequency and effort (RQ_1 and RQ_2, respectively). In other words, this dissertation investigates if modules with high levels of TD amount present more frequent and more intense corrective maintenance activities, compared to modules with lower TD amounts. It should be noted that in order to answer RQ_2, it was needed to transform [V3] from a continuous to a binary variable. As low (high) TD files are characterized the ones that have technical debt that falls below (higher than) the median TD amount across all files for that project.

The aforementioned tests are fitting ways to assess the consistency/correlation and discriminative power of metrics, as described by 1061:1998 IEEE Standard for Software Quality Metrics[15].

3. Results

Table 3 lists the results of the conducted Spearman's correlation analysis for each project for both RQs. Concerning RQ_1, in all ten projects there is a statistically significant positive correlation between TD amount of a file and the number of times that file underwent corrective maintenance (***interest probability***). Regarding RQ_2, in 8 out of 10 projects there is a statistically significant positive correlation between the amount of TD of a file and the extent of modification that the file underwent during corrective maintenance (related to ***interest amount***).

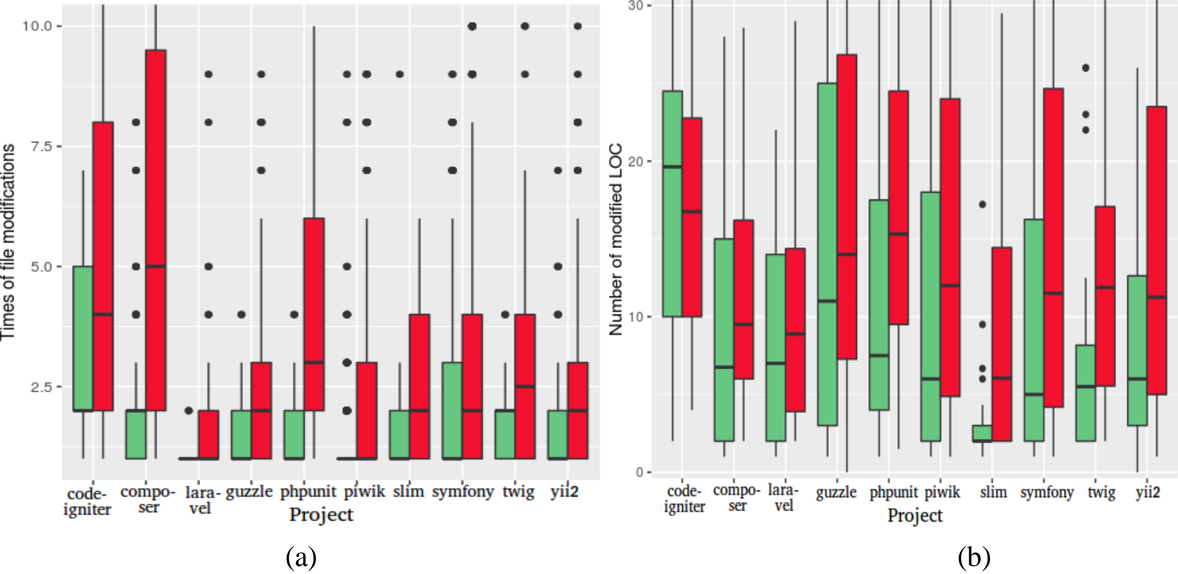
[14] https://github.com/theoAm/githubGrabber

[15] 1061-1998 IEEE Standard for a Software Quality Metrics Methodology, IEEE Standards, IEEE Computer Society, 31 December 1998 (re-affirmed 9 December 2009).

Table 3. Spearman's Correlation Results

project	RQ1		RQ2	
	p	r	p	r
CodeIgniter	0.00	0.293	0.08	-0.124
Symfony	0.00	0.301	0.00	0.280
Composer	0.00	0.544	0.00	0.310
Yii2	0.00	0.278	0.00	0.262
Guzzle	0.00	0.366	0.01	0.178
Slim	0.00	0.409	0.00	0.591
Laravel (kernel)	0.00	0.481	0.17	0.148
Piwik	0.00	0.363	0.00	0.204
PHPunit	0.00	0.626	0.00	0.290
Twig	0.00	0.366	0.00	0.433

To allow a visual interpretation of the results, Figure 2 depicts the two indicators of the required effort (times that a file undergoes defect-related changes and the extent of changes in terms of lines of code) for each project, by differentiating between low and high TD files. As it becomes evident from the box plots, the required maintenance is always (except for one case in Figure 2(b)) larger for high TD modules. This finding is also supported by the results of the Mann-Whitney U test which suggest that [V2] and [V3] in high-TD files are statistically different from [V2] and [V3] in low-TD files ([V2]: p-value = ~0.00, [V3]: p-value = ~0.00). On average, the number of times that a high TD file is modified is **1.9** times larger than the number of times a low TD file is changed. In terms of the extent of change, the corresponding ratio is **2.4** to 1.



(a) (b)

Figure 2. Discriminative power of TD amount (left/right bars correspond to low/high TD files, respectively)

4. Threats to Validity

The results of the study are subject to external validity threats since the investigation has been performed on 10 PHP projects. Further studies on other projects or languages would be valuable in assessing the relation between TD amount and interest probability/amount in different contexts. Moreover, the assessment of interest amount through the extent of modification poses a threat to construct validity, since interest should be ideally quantified as the difference between the nominal effort for fixing an issue (i.e. in case no TD were present) and the actual effort spent. The former effort is unfortunately unknown. However, the findings observed when high TD modules are contrasted to low TD ones, imply

that increased frequency and extent of modification are often encountered in files with increased interest amount.

A second threat to construct validity stems from the fact that not all reported issues point to errors, but some of them might contain a feature request or suggestion for performance improvement. As a result, any actions to handle this issue would constitute adaptive or perfective maintenance rather than corrective one. Another threat of the same category, is that bug-related commits, which indeed fix an issue, but do not employ the keywords sought by GitHub, will be missed. This threat implies that there might be other bug-related commits which have been neglected in the study.

A final threat pertaining to the construct validity of the study stems from the fact that TD amount and the two employed indicators of corrective maintenance are aggregated over multiple revisions, possibly accounting for a significant period of time. As a result, especially in the case of variations of TD or corrective maintenance during that time, it cannot be safely assumed that the measured levels of corrective maintenance correspond to the measured TD. For example, an observed high level of corrective maintenance in a module with high level of TD, could in fact be due to a particular sub-period in which the module had low TD.

Finally, the present study does not investigate whether the two interest-related variables of the research questions (i.e., *frequency of modifications* and *extent of modification* due to corrective maintenance) might be affected by the propagation of errors. In particular, the study focuses on the relation between the two aforementioned variables and the TD principal of the files in which errors have been fixed, possibly neglecting the TD of the originating files (i.e., those from which errors might have propagated). This treatment poses a threat to construct validity and constitutes an interesting research direction for future work.

5. Discussion and Conclusions

The results of this chapter suggest that TD amount is indeed correlated with maintenance effort. In particular, developers appear to spend more time on fixing issues in files with high levels of accrued technical debt, compared to files that present less TD. Therefore, project managers should take quality-oriented decisions to deter the appearance of software units with increased technical debt.

With respect to practitioners, the results provide additional evidence that TD undermines software maintenance and that it should be taken under consideration before any design and implementation decision. Moreover, the domain of the study suggests that TD appears to be important in a web context as well. Software engineers can take advantage of such empirical evidence to convince management about the importance and need to manage TD.

From a research perspective, since there is sufficient empirical evidence of the impact of TD amount on corrective maintenance, the need to devise a framework for assessing the associated risk and costs of managing TD becomes essential.

References

[1] N. S. R. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman, “Identification and management of technical debt: A systematic mapping study,” *Inf. Softw. Technol.*, vol. 70, pp. 100–121, Feb. 2016.
[2] A. Ampatzoglou, A. Ampatzoglou, P. Avgeriou, and A. Chatzigeorgiou, “A Financial Approach for Managing Interest in Technical Debt,” in *Business Modeling and Software Design*, B. Shishkov, Ed. Springer International Publishing, 2015, pp. 117–133.
[3] A. MacCormack and D. J. Sturtevant, “Technical debt and system architecture: The impact of coupling on defect-related activity,” *J. Syst. Softw.*, vol. 120, pp. 170–182, Oct. 2016.

[4] Q. Feng, R. Kazman, Y. Cai, R. Mo, and L. Xiao, “Towards an Architecture-Centric Approach to Security Analysis,” in *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2016, pp. 221–230.

[5] R. L. Nord, I. Ozkaya, E. J. Schwartz, F. Shull, and R. Kazman, “Can Knowledge of Technical Debt Help Identify Software Vulnerabilities?,” presented at the 9th Workshop on Cyber Security Experimentation and Test (CSET 16), 2016.

[6] N. Zazworka, A. Vetrò, C. Izurieta, S. Wong, Y. Cai, C. Seaman, and F. Shull, “Comparing four approaches for technical debt identification,” *Softw. Qual. J.*, vol. 22, no. 3, pp. 403–426, Apr. 2013.

[7] Z. Li, P. Liang, P. Avgeriou, N. Guelfi, and A. Ampatzoglou, “An Empirical Investigation of Modularity Metrics for Indicating Architectural Technical Debt,” in *Proceedings of the 10th International ACM Sigsoft Conference on Quality of Software Architectures*, New York, NY, USA, 2014, pp. 119–128.

[8] R. Schwanke, L. Xiao, and Y. Cai, “Measuring Architecture Quality by Structure Plus History Analysis,” in *Proceedings of the 2013 International Conference on Software Engineering*, Piscataway, USA, 2013, pp. 891–900.

[9] G. A. Oliva, I. Steinmacher, I. Wiese, and M. A. Gerosa, “What Can Commit Metadata Tell Us About Design Degradation?,” in *Proceedings of the 2013 International Workshop on Principles of Software Evolution*, New York, NY, USA, 2013, pp. 18–27.

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

Chapter V. PERSONALIZED ASSESSMENT OF TECHNICAL DEBT PRINCIPAL

The work of this chapter was published in the Proceedings of the XP2017 Scientific Workshops, MTD2017:

Amanatidis, Theodoros & Chatzigeorgiou, Alexander & Ampatzoglou, Apostolos & Stamelos, Ioannis. (2017). Who is Producing More Technical Debt? A Personalized Assessment of TD Principal. 10.1145/3120459.3120464.

Chapter Summary

Technical debt (TD) impedes software projects by reducing the velocity of development teams during software evolution. Although TD is usually assessed on either the entire system or on individual software artifacts, it is the actual craftsmanship of developers that causes the accumulation of TD. In the light of extremely high maintenance cost, efficient software project management cannot occur without recognizing the relation between developer characteristics and the tendency to evoke violations that lead to TD. This chapter investigates three research questions related to the distribution of TD among the developers of a software project, the types of violations caused by each developer and the relation between developers' maturity and the tendency to accumulate TD. The study has been performed on four widely employed PHP open-source projects. All developers' personal characteristics have been anonymized.

1. Introduction

Tom DeMarco in his novel about project management ("The Deadline") [1] vividly claims that the most important part of any successful software project is team and people. According to Mr. Tompkins, the main character of the story, *people do projects* and therefore *getting the right people* is essential. Different developers have varying skills and capabilities in designing, developing and maintaining software in the right manner. Unavoidably, the members of a development team introduce design and code violations at unequal rates and intensities, contributing differently to the overall system Technical Debt [2].

Technical Debt principal (i.e., the effort needed to refactor a system in order to address existing inefficiencies) is usually assessed on design or code artifacts. However, since software development is a highly people-centric activity, Technical Debt Management (TDM) should also consider the individual members of a team. To name an example, technical debt items with high interest probability [3] (i.e. modules that hold TD and are very likely to undergo maintenance in the future) should be assigned to skilled and experienced developers to mitigate the involved risks.

Acknowledging that efficient project management cannot take place unless people are carefully matched to tasks, this chapter presents the results of a case study assessing the distribution of TD among developers. Knowing whether some members of the development team are more likely to introduce TD or particular design/code violations can be of value to project managers to steer the allocation of issues and maintenance tasks more effectively. Moreover, it is also investigated whether the tendency to introduce TD is related to the developer's age in the project. The relevant research questions have been investigated based on findings from four widely employed PHP open-source projects with a long development history.

Collecting and processing information at the level of individual developers involves a number of ethical issues and therefore should be performed with care. All gathered personal data, which are subject to statistical analysis, has been de-identified. In any case, assessing the contribution of the members of a development team to the system's TD for research purposes, should not share any kind of personal data with third parties. On the other hand, performance appraisals within an organization are a great and commonly used tool to evaluate how employees have been performing. It should be noted however, that any type of performance analysis should respect ethics, ensuring for example that developers are aware of the relevant process and that any feedback will be accessible by the employees and will remain confidential.

The rest of the chapter is organized as follows: Section 2 provides an overview of related work on the assessment of software quality at the developer level, regardless of whether TD is explicitly mentioned or not. The case study design is presented in Section 3 while the results for each of the investigated questions are presented and discussed in Section 4. Implications to project managers and developers are presented in Section 5, while threats to the validity of the study are discussed in Section 6. Finally, conclusions are drawn in Section 7.

2. Related Work

This section presents efforts that aimed at investigating how the characteristics and coding habits of individual developers relate to the introduction of code smells, violations and buggy code that eventually undermine software quality.

Alves et al. investigated the influence of developers on the introduction of code smells in 5 open source software systems [4]. Developers have been classified in different groups based on two characteristics, namely: a) developer participation, calculated as the time interval between his first and last commit and b) developer authorship, representing the number of modified files and lines of code. The authors investigated how those two characteristics are related to the insertion and/or removal of five types of code smells: dead (unused) code, large classes, long methods, long parameter list (of methods) and unhandled exceptions. Results suggested that groups with fewer participation in code development tended to have a greater engagement in the introduction and removal of code smells. Authors supported that groups with higher participation level code more responsibly during maintenance whereas the other groups tend to focus on error correction actions.

Tufano et al. analyzed developer-related factors, on 5 open source Java projects, that could influence the likelihood of a commit to induce a fix [5]. They found evidence that clean commits (i.e., commits that do not induce bugs or any kind of need to fix code) have higher coherence than fix-inducing commits. Commits with changes that are focused on a specific topic or subsystem are considered more coherent than those with more scattered changes. Furthermore, their results, surprisingly, suggested that developers with higher experience perform more fix-inducing commits that developers with lower experience. Authors claimed that this could be happening due to the fact that more experienced developers usually cope with more pretentious tasks.

Eyolfson et al. [6] analyzed the impact of three social characteristics of commits on their bugginess: a) time of the day the commit is performed, b) day of the week, and c) developer's experience (i.e. days of participation in the project) and commit frequency. The study was performed on two open source projects (the Linux kernel and PostgreSQL) and found evidence that late-night commits are significantly buggier emphasizing that developers that perform late-night commits should double-check their code. They also found that more experienced developers introduce fewer bugs. Furthermore, according to their results, the day on which the code is written plays no significant role on the 'bugginess' of a commit something which contradicts what was observed in an earlier study by Sliwerski et al. back in 2005 [7]. That study claimed that programming on Friday is more likely to generate faults than on any other day.

Rahman and Devanbu [8] studied the impact of ownership and experience of the developers on the quality of code. As ownership, they considered the extent to which a developer modifies a file along with others or on his own. They also conceptualized two distinct types of experience that can affect the quality of a developer's work: specialized experience in a file (i.e. developer's contribution to a single file) and general experience in the entire project (i.e., developer's contribution to the entire project). Their results highlighted that: a) code that is maintained by many developers is less bug-prone, validating the "many eyeballs → better code" theory, b) less specialized experience on a specific file is associated with fix-inducing code to that file and c) the lack of general experience on the overall project is not consistently associated with faulty code.

This dissertation differs in that software quality is viewed from the perspective of TD rather than the introduction of faults or selected code smells. Although not all TD violations are considered as harmful by development teams, examining a broader range of design and code inefficiencies as well as the distribution of TD introduction among developers can provide a more holistic view on the competencies of a team.

3. Case Study Design

3.1. Research Objectives and Research Questions

The aim of this chapter, expressed through a GQM formulation, is: to **analyze** individual contributions by the project developers **for the purpose of** evaluation **with respect to** the TD that they introduce, **from the point of view of** software managers **in the context of** software maintenance and evolution in open-source projects.

Driven by this goal, three relevant research questions have been set:

RQ_1: *Is TD uniformly distributed among the developers of a software project?*

The first research question aims to investigate whether TD is uniformly induced by all developers in a software project or is mostly associated to the commits of specific developers. Answering this research question and especially if common patterns among the examined projects are found, could shed light into the actual causes of design and code inefficiencies.

RQ_2: *Which TD violations are introduced by the developers of a software project?*

The second research question concerns the particular TD violations caused by each developer during his commits and investigates whether there is any relation between violation types and developers. Any evidence on commonly occurring violations across all developers or individual members of the development team can be of help to efficient technical management.

RQ_3: *What is the relation between TD and the maturity of developers in a software project?*

The third research question analyzes the relation between the maturity of each developer in any project (obtained as the time since his initial commit to the project) and his tendency of inducing TD. It would be reasonable to assume that less experienced developers introduce more TD and thus allocation of work considering the maturity factor would enable effective TD management.

3.2. Case and Units of Analysis

This is an embedded multiple-case study, i.e. it studies multiple cases, whereas each case is comprised of many units of analysis. Specifically, the cases of the study are open source projects, and units of analysis are the developers of each project. The reporting of results is performed at the project/case level.

As subjects for the study, recent commits (i.e. those of the most recent year) of a selected branch during the development history of 4 open source projects written in PHP, were obtained. The projects have been selected so as to have a long development history and varying sizes. A short description of the goals of these projects is provided below, whereas some demographics are provided in Table 1. ***Laravel* (core)** consists of the core source code of one of the most popular PHP frameworks for building web applications, Laravel, with more than 20 million downloads. ***Composer*** is the most popular dependency manager for PHP with more than 2 million downloads. ***Yii2*** and ***CakePHP*** are two actively maintained PHP frameworks with over 2.5 million and 1 million downloads respectively.

All developers who submitted at least 10 commits on the examined branches of the selected projects have been used as cases for this dissertation (the lower limit of 10 commits has been set to avoid considering in the study developers with partial or circumstantial association to the project).

Table 1. OSS PHP Project Demographics

Project	**#Commits**	**#Developers (considered)**	**Size of last version (LOC)**
Laravel (core)	1136	11	149K
Composer	807	7	8K
Yii2	2097	19	406K
Cakephp	1677	23	297K

3.3. Variables and Data Collection

3.3.1. Variables

For each unit of analysis (i.e. developer in a project) and in order to answer the research questions that have been set, the following variables were recorded:

[V1] **DevID**: unique developer identification id

[V2] **Total TD**: induced TD by all commits of the particular developer during the examined time frame. Contributed TD for a particular transition from one commit to the next is obtained by SonarQube as the difference between the TD of the files that the developer modified during the transition. It can be positive or negative.

[V3] **Number of modified lines**: To normalize the contributed TD over the amount of work performed by each developer the number of lines that have been modified during each commit was recorded (as the number of added and deleted lines of code).

[V4] **Normalized TD**: Since the amount of TD that is introduced by a developer is heavily dependent on the amount of code that he contributes, to allow for a fair assessment the total TD (V[2]) is normalized by dividing it with the number of modified lines (V[3]).

[V5] **Types of TD violations**: This variable consists in a map of TD violation types and occurrence frequencies. It essentially captures the types of TD violations caused by the commits of each developer.

[V6] **Developer Maturity**: Time between the first commit that each developer performed in the project's history to the last commit that he contributed. It captures the developer's maturity in the project.

3.3.2. Data Collection

In order to analyze developers' recent activity and contribution to Technical Debt the most recent year's commit data for every examined project was obtained, via the GitHub API. This data includes commit information, such as the author of the commit, the number of changed lines of code, the modified files, the commit date and of course the commit id (hash) in the repository. Next, the TD of every project snapshot, corresponding to each commit, has been calculated using SonarQube[16]. SonarQube is a widely employed tool for assessing technical debt that quantifies the principal based on several axes of code quality (e.g., code duplications, metrics, styling conventions, etc.). In particular, the source code corresponding to each commit was iterated and performed TD analysis with SonarQube for every project snapshot. The entire process has been fully automated by executing the required commands within a bash script.

Once the analysis for each project snapshot has been completed, commits have been grouped by developers and placed in chronological order. For every developer's commit, the files that he/she modified have been identified, and their TD amount has been compared against the TD of the same files in the previous commit[17] that involved those files. The difference in TD amount that was detected between two successive commits (ignoring the commits affecting other files) was added to each developer's stack and eventually calculated the total contribution of each developer to the project's technical debt principal. The process of obtaining the personalized principal contribution (delta of TD) based on two successive commits is illustrated in Figure 1.

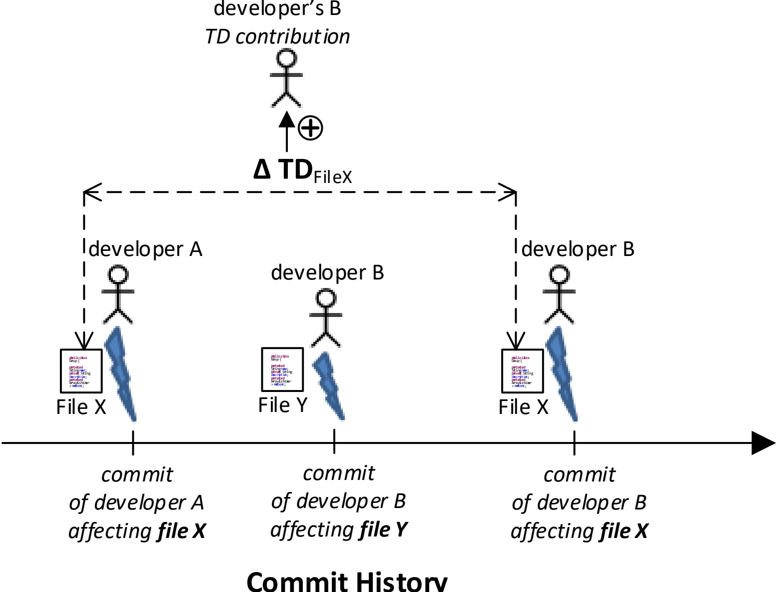


Figure 1. Process of obtaining TD deltas for each developer

3.4. Data Analysis

To answer the research questions stated in Section 3.1, using the variables described in Section 3.3, descriptive statistics and hypothesis testing (for RQ_3) were employed.

For checking whether the distribution of TD among developers is uniform or not (RQ_1), the distribution will be presented as a bar chart. To provide a more systematic view into the distribution of TD, the Gini

[16] Available at: http://www.sonarqube.org

[17] For the special case where a file was created in a particular commit and thus did not exist in the previous commit, zero TD principal has been assumed for the previous commit

coefficient was calculated for each project. The Gini coefficient is a measure of statistical dispersion originally used for quantifying the inequality of income distribution [9]. The value of the Gini coefficient varies between zero and one. A Gini coefficient (or index) equal to zero implies perfect equality in the distribution (i.e. the case where all developers introduced the same amount of TD). A Gini index equal to one, implies maximum inequality (i.e. the case where one developer introduces the entire TD of the system while all others introduce no TD at all).

To investigate whether developers have a tendency to introduce particular TD violations (RQ_2) a heatmap was used. Columns correspond to the individual developers in each project (denoted by their ID) while rows correspond to identified TD violations as obtained by SonarQube. Frequently occurring violations are denoted by darker colors. A completely black cell indicates that the corresponding developer introduces only violations of one type (that corresponding to the row). In case the violations by a developer are distributed among many types, shading changes according to the percentage of violations of each type.

Finally, to test whether developer maturity plays a role in the number and severity of violations that they introduced, the findings are displayed as scatterplots (developer age vs. normalized TD) and the hypothesis whether normalized TD depends on age is tested with correlation analysis. Since correlation analysis on the limited data points of each project leads to statistically insignificant results, for this research question a combined dataset from all projects has been formed. However, to avoid any biasing, the combined dataset contains developer maturity and introduced normalized TD expressed as a percentage: For each project, the maturity of each developer (in days) is divided with the maturity of the most experienced developer. Similarly, for each project, the normalized TD (i.e. TD/LOC) for each developer, is divided by the maximum normalized TD in that project.

To further investigate whether developer's maturity is related to the amount of introduced TD principal, an independent study t-test has been performed, by differentiating between less and more experienced developers (the age in days corresponding to 50% of the longest experience was used as threshold). The analysis strategy per research question is summarized in Table 2.

Table 2. Data Analysis

RQ	Analysis Strategy
RQ_1	Bar-chart illustrating distribution of TD [V4] among developers [V1] – Gini index for each distribution
RQ_2	Heatmap illustrating frequency and types of violations [V5] per developer [V1]
RQ_3	Scatterplot & correlation analysis between normalized TD [V4] and developer age [V6] Independent sample t-test, grouping variable [V6] (threshold 50%) and testing variable [V4]

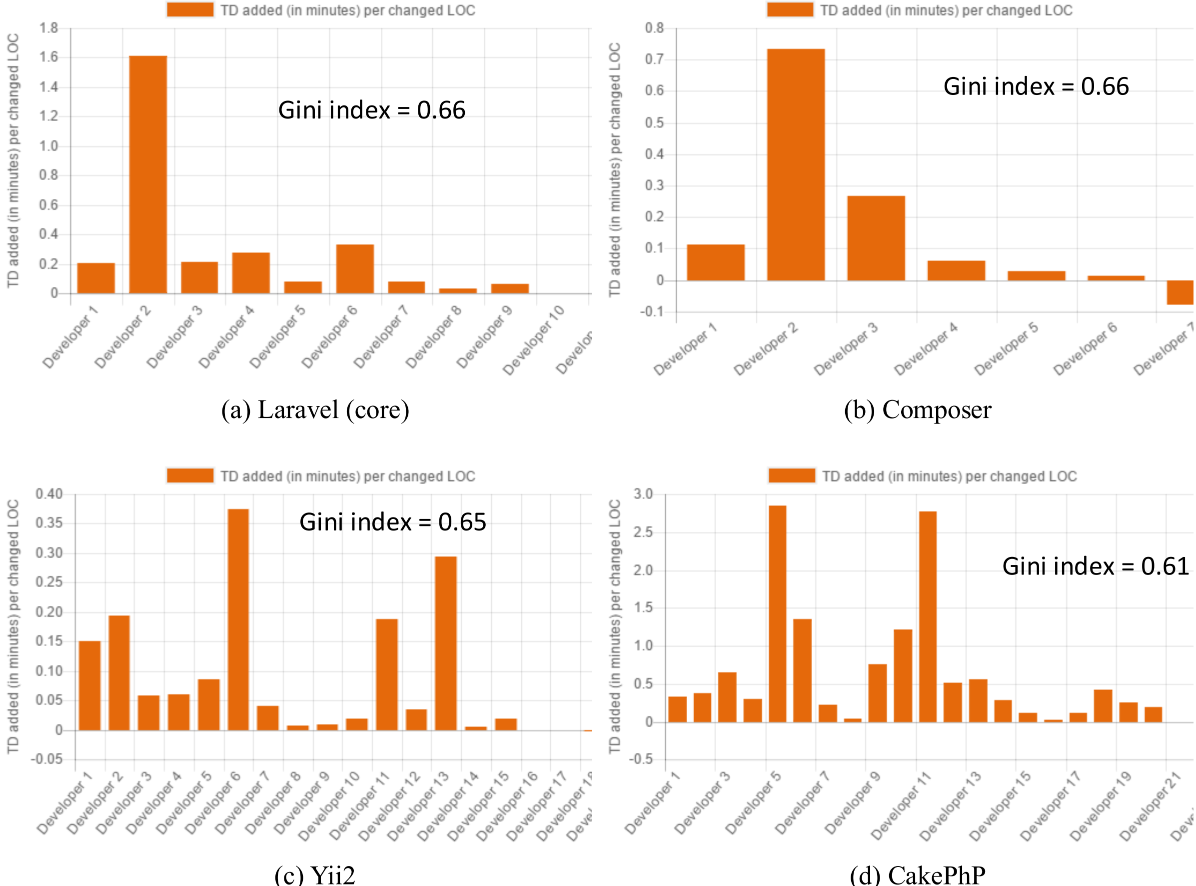
4. Results and Discussion

This section presents the results of the study organized per research question along with an interpretation of the findings.

4.1. Distribution of TD among Developers

Figure 2 illustrates the distribution of the contributed TD during the examined time frame among the developers who performed commits in each project. To avoid biasing the results by the amount of code

written by each developer and thus 'falsely blaming' a developer, the added TD is normalized over the number of changed lines of code. On each chart the value of the corresponding Gini index is also shown.



(a) Laravel (core)

(b) Composer

(c) Yii2

(d) CakePhP

Figure 2. Distribution of TD among developers

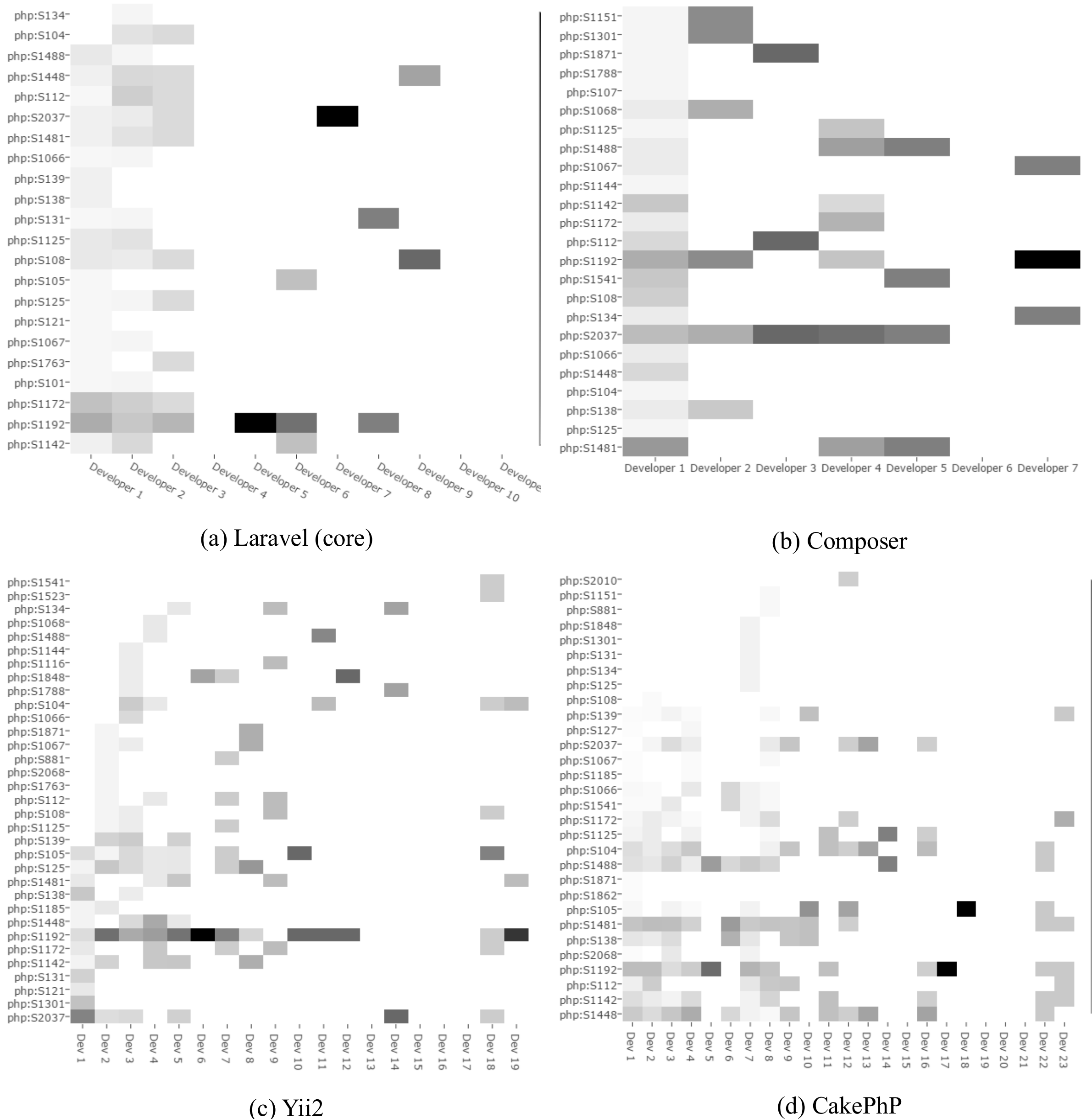
The pattern observed in each plot presents similarities across projects. A limited number of developers (e.g. *Developer-2 for Laravel* and *Developer-5 and Developer-11 for CakePHP*) contribute a significant portion of the system's technical debt (in terms of TD per line of code), while the majority of developers contribute significantly less violations. In a few cases developers even have a negative TD contribution meaning that they remove violations instead of introducing new ones when adding code.

The distribution in general is far from uniform as it is confirmed by the Gini index which is remarkably similar in all projects. To provide an intuitive interpretation of the meaning of the Gini index, it is noted that a Gini value of 0.66 implies that 80% of the developers introduce approximately 1/3 of the system's TD. The rest 2/3 is introduced by only 20% of the developers. Therefore, there is a small group of developers that produce significant amount of principal, whereas another larger set of developers produces less technical debt confirming the Pareto principle.

It could be claimed that TD principal is not equally distributed across developers since at least one of them stands up as a main source of producing violations (and therefore introducing principal). On the contrary, there are cases in which developers consistently remove violations (i.e., repay TD). However, this observation is not consistent across all investigated projects

4.2. TD Violations per Developer

Figure 3 illustrates the most common violations in each of the examined projects against the developers who introduce them, in the form of a heatmap. The darker the color the more violations of the corresponding type are introduced by the indicated developer. A row that is relatively dark across all developers implies a commonly occurring violation. On the other hand, a column with many dark cells implies a developer that generates many different types of violations.



(a) Laravel (core)

(b) Composer

(c) Yii2

(d) CakePhP

Figure 3. TD violation types per developer

The findings vary among projects, similarly to the total number of different violation types encountered in each project (22 violation types in Laravel to 30 types in CakePhP). Rows with many shaded cells indicate common violation types introduced by many developers. Such a violation is violation '`php:S1192`' (of critical importance) in all projects. According to SonarQube this violation indicates the presence of String literals which are duplicated, rendering the process of updating all occurrences in case of a change, error-prone. Another relatively common violation among developers in all projects is

'`php: S2037`' (of minor importance). SonarQube identifies as violations cases where a reference to a static class member from another method in the same class is not employing the "`static::`" keyword. This might lead to undesired behavior in the case of subclasses, as the original definition of the member is referenced, rather than the overridden one.

Differences are also clearly visible between developers. Some developers introduce violations of many different types, as indicated by shaded cells in the corresponding columns. This is for example the case for the first three developers of project Laravel. In such cases, training actions focusing on the merits of smell-free code can be planned as part of a project's management for selected members of the development team. On the other hand, some developers produce violations of a very limited number of types, even of a single type. This is for example the case for developers with a single black cell in their column (i.e. 100% of their violations belong to that specific type). Although the latter information might be of limited value to a project manager, it could be useful as a self-assessment tool for the developer. The analysis points to the particular violations that a developer is inclined to introduce, and if he acknowledges their importance, can eventually modify his programming habits to eliminate them.

In principal a large variety of violations can be identified in different projects, introduced by different developers. However, this dissertation points out specific frequently recurring violations for: (a) the same project, (b) the same developer, and (c) across all projects.

4.3. TD vs. Developer Maturity

The third research question aims at investigating the relation between a developer's 'age' in the project and the TD that he introduced per line of code. The corresponding scatterplot for variables [V4] and [V6] is shown in Figure 4. The trendline in the chart indicates a very moderate negative correlation between developer maturity and introduced TD (note that both variables are expressed as ratio over the highest developer maturity and the highest TD/LOC in each project, respectively). However, the p-value for Spearman correlation indicates that the results are not statistically significant ($p = 0.753$). Thus, there is no evidence to support the rejection of the corresponding null hypothesis (i.e. that no monotonic correlation between the two variables exist).

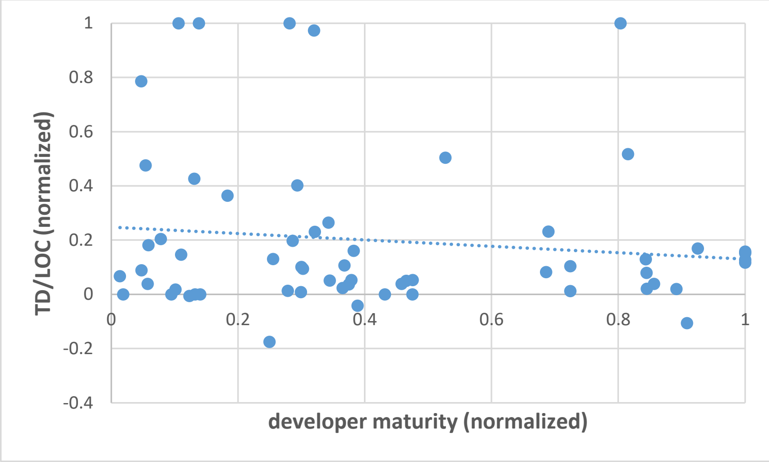


Figure 4. Introduced TD versus developer maturity

To further investigate whether developer's maturity plays any role in the amount of introduced TD principal an independent study t-test has been performed. However, the results of the test have not suggested the rejection of the null hypothesis (sig: 0.8). Therefore, one cannot claim that there is a difference in the mean TD incurred by experienced and inexperienced software developers.

However, despite the lack of statistical evidence it can be observed that a larger number of immature developers is concentrated in the top-20% most TD-incurring developers (5 immatures against 1

experienced). This finding, in conjunction with the declining trendline in the scatterplot opens up an interesting research direction. In particular, the identification of additional factors (apart from experience) that characterize the developer need to be investigated so as to more accurately profile which types of developers incur the most TD principal.

The collected data were not able to provide enough evidence on the relationship between developers' age and the amount of TD that they introduce. However, a negative trendline has been identified and 80% of the most TD-introducing developers have been active for less than 33% of the project's age (i.e., have low project-related experience).

5. Implications of the Study

Any performance analysis at the level of individual people might be viewed with skepticism. However, the provided perspective on a system's TD and its actual causes might prove beneficial to the managers of software development teams and to the developers themselves.

With respect to software project managers, resource allocation can benefit by assigning artifacts with increased technical debt interest probability to software engineers that tend to introduce less technical debt principal or even remove technical debt. In a similar line of thought, and without any intent to punish developers, managers could identify developers who impair software quality by introducing source code violations and technical debt instances and try to upgrade their coding habits, either by placing them next to more experienced developers or by calling them to reflect on their common violations. Appropriate guidelines or tooling to avoid the accumulation of particular violations can also be developed, based on the findings from previous projects.

With respect to software developers, the results on the personalized assessment of technical debt can be a valuable self-improvement tool. Developers can identify recurring problems that they consciously or unconsciously introduce as well as their locations in code. Moreover, critically analyzing their own performance with respect to TD against the rest members of their team can highlight opportunities for improvement.

Finally, the results of the study provide some useful research implications as well. First, the outcomes of the study suggest that an individual / personalized assessment of TD can be a meaningful research direction that unveils interesting relations that can guide TDM. Therefore, the topic deserves further investigation. Some tentative future research direction are as follows: (a) a personalized assessment of TD interest, (b) a detailed analysis of specific violations, with respect to their criticality, and (c) an elaborate personality / developers' characteristics model that will provide a more accurate profile of TD-prone developers.

6. Threats to Validity

This section presents and discusses threats to the validity of the empirical study emphasizing on construct, reliability, external and internal validity threats, according to the classification by Runeson et al. [10].

Construct validity reflects to what extent the phenomenon under study (i.e. introduction of technical debt principal by individual developers) really represents what is investigated according to the research questions. By selecting a particular tool for quantifying technical debt, whereas other types of non-identified technical debt exist, threats to construct validity emerge. However, SonarQube is a widely employed tool for the assessment of technical debt identifying a variety of design and code inefficiencies.

The reliability of a case study is related to the extent by which the collected information and the performed analysis can be replicated with the same results. To mitigate reliability threats, the design of the case study and the statistical tests that have been performed are explicitly reported.

Internal validity threats are related to the identification of confounding factors, that is, variables, other than the implied independent variables (developer's competence and maturity) which might influence the value of the dependent variable (introduced technical debt and technical debt types). Such threats do apply in the presented study, since introduced technical debt might be affected by the tasks assigned to (or chosen by) each developer. For example, a highly skilled and experienced developer might be inclined to take over the most complex and demanding tasks limiting his ability to control the introduced technical debt.

Finally, as in any other empirical study, the results are subject to external validity threats. External validity deals with the possibility to generalize the findings. To mitigate this threat, four widely known PHP projects have been selected, which have evolved over a number of years. Nevertheless, further studies are required to thoroughly analyze the parameters that drive developers to introduce TD.

7. Conclusions

Software development is a complex activity requiring experience, skills and significant mental effort. Artifacts produced by developers are systematically analyzed in terms of quality, which recently is successfully captured by the Technical Debt metaphor. In this chapter, the relation between introduced TD principal and developers has been investigated, through a case study on four open-source PHP projects.

The findings confirm the belief that developers' competencies vary, since the distribution of technical debt among developers is highly imbalanced. Moreover, different developers introduce different technical debt violations; however, some recurring violations can be identified across developers and projects. Finally, there is no statistically significant evidence that more experienced developers introduce less technical debt per line of code. Such findings but more importantly the ability to perform a personalized assessment of technical debt can be a valuable tool for effective project management and self-assessment and improvement.

References

[1] T. DeMarco, *The Deadline: A Novel About Project Management*. New York: Computer Bookshops, 1997.

[2] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical Debt: From Metaphor to Theory and Practice," *IEEE Softw.*, vol. 29, no. 6, pp. 18–21, Nov. 2012.

[3] N. S. R. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman, "Identification and management of technical debt: A systematic mapping study," *Inf. Softw. Technol.*, vol. 70, pp. 100–121, Feb. 2016.

[4] L. Alves, R. Choren, and E. Alves, "An Exploratory Study on the Influence of Developers in Code Smell Introduction," in *Proceedings of the 10th International Conference on Software Engineering Advances* (ICSEA 2015), Barcelona, Spain, 2015.

[5] M. Tufano, G. Bavota, D. Poshyvanyk, M. Di Penta, R. Oliveto, and A. De Lucia, "An empirical study on developer-related factors characterizing fix-inducing commits," *J. Softw. Evol. Process*, vol. 29, no. 1, Jan. 2017.

[6] J. Eyolfson, L. Tan, and P. Lam, "Do Time of Day and Developer Experience Affect Commit Bugginess?," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, New York, NY, USA, 2011, pp. 153–162.

[7] J. Sliwerski, T. Zimmermann, and A. Zeller, "Don't Program on Fridays! How to Locate Fix-Inducing Changes," in *Proceedings of the 7th Workshop on Software Reengineering*, Bad Honnef, Germany, 2005.

[8] F. Rahman and P. Devanbu, "Ownership, experience and defects: a fine-grained study of authorship," n *Proceedings of the* 33rd *International Conference on Software Engineering*, Waikiki, Honolulu, USA, 2011, p. 491.

[9] C. Gini, "Measurement of Inequality of Incomes," *Econ. J.*, vol. 31, no. 121, pp. 124–126, 1921.

[10] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case Study Research in Software Engineering: Guidelines and Examples*, 1st ed. Wiley Publishing, 2012.

Chapter VI. FACTORS AFFECTING DECISION TO REPAY TECHNICAL DEBT

The work of this chapter was published in the Proceedings of the 2018 International Conference on Technical Debt, TechDebt 2018 (Pages 62-66):

Amanatidis, Theodoros & Mittas, Nikolaos & Chatzigeorgiou, Alexander & Ampatzoglou, Apostolos & Angelis, Lefteris. (2018). The developer's dilemma: Factors affecting the Decision to Repay Code Debt. 10.1145/3194164.3194174.

Chapter Summary

The set of concepts collectively known as Technical Debt (TD) assume that software liabilities set up a context that can make a future change more costly or impossible and therefore repaying the debt should be pursued. However, software developers often disagree with an automatically generated list of improvement suggestions, which they consider not fitting or important for their own code. To shed light into the reasons that drive developers to adopt or reject refactoring opportunities (i.e. TD repayment), in this chapter, an empirical study on the potential factors that affect the developers' decision to agree with the removal of a specific TD liability, is performed. The study has been addressed to the developers of four well-known open-source applications. To increase the response rate, a personalized assessment has first been sent to each developer, summarizing his/her own contribution to the TD of the corresponding project. Responds have been collected through a custom-built web application that presented code fragments suffering from violations as identified by SonarQube along with information that could possibly affect their level of agreement to the importance of resolving an issue. These factors include data such as the frequency of past changes in the module under study, the number of bugs, the type and intensity of the violation, the level of involvement of the developer and whether he/she is a contributor in the corresponding project. Multivariate statistical analysis methods have been used to understand the importance and the underlying relationships among these factors and the results are expected to be useful for researchers and practitioners in TD Management.

1. Introduction

According to A. Hunt and D. Thomas many developers are reluctant to start 'ripping up' their code (a.k.a. refactor) just because it isn't quite right [1]. As they vividly put it, going to a boss or client and saying that a working piece of code needs another week to refactor it, would probably cause a response that cannot be printed. However, deferring a refactoring might incur technical debt (TD) requiring greater time investment to fix the problem down the road.

Previous studies have shown that developers perceive and handle TD in different ways [2] and have distinct motivations for applying refactorings [3]. To shed light into the factors that drive developers to accept or reject automated suggestions for TD removal, this dissertation targets at developers of open-source PHP projects with the following two main characteristics: (a) to increase their motivation for participating in the study each participant was provided, prior to requesting his feedback, with a personalized report on the TD that he/she has incurred to the project, and (b) to facilitate the collection of data a web application has been implemented to present individual code fragments suffering from an identified TD issue along with information on the parameters that might affect the developers decision to repay the TD or not.

2. Related Work

There is a limited number of studies providing insights on the developer's perception regarding the urgency to resolve code violations, which in turn lead to the accumulation of TD. In a recent study [4], the authors sent surveys to explore whether issues involving architectural elements lie among the most significant sources of TD. A number of 536 respondents replied (leading to a response rate of 29%) and the results showed that architectural issues are the greatest source of TD. Such issues are difficult to cope with and they dragged on for many years. Another explored subject was the existence of effective tools for managing TD. Respondents claimed that existing tools do not capture the key areas of accumulated problems related to TD.

In a 2014 study [5], the authors investigated which bad smells are considered by the developers as the most harmful. The developers were given code snippets from three systems with twelve kinds of bad smells and were asked to rank the severity of the smells. Both original developers from the systems and outsiders (industrial developers) were included in the survey (a response rate of 40% was achieved). The results suggested that smells related to complex code are considered an important threat by developers.

In another exploratory study [6], comments of four large open-source systems were used to identify self-admitted TD. The authors found that more experienced developers introduce most of the self-admitted TD while time pressure and code complexity do not relate to the amount of self-admitted debt.

In another study [7], 20 developers were interviewed to investigate why static analysis tools are not used during development. Participants claimed that static analysis tools are beneficial, but false positives, poor output and low customizability deter their use. Spinola et al. [8] chose 14 statements regarding TD and asked 37 practitioners if they agree with them. The statement "Not all technical debt is bad" lies among those with the maximum consensus. In other words, developers believe that there is a healthy level of TD in every system.

Kim et al. [9] conducted a survey to examine developers' perception regarding code refactoring. Participants responded that refactoring hides substantial cost and risks and further support is needed beyond automated refactoring within IDEs. However, a case study on Windows 7 highlighted the benefits of refactoring. The results showed that "*refactored modules experienced higher reduction in the number of inter-module dependencies and post-release defects than other changed modules*".

In another study which debated developers' position about code refactoring [10], 20 refactoring practitioners were interviewed. Participants recognized the added value of a refactoring (code reusability), however if too much effort is needed, they may be reluctant to make refactoring decisions. Mäntylä and Lassenius [11] studied the refactoring decisions made by 37 students on a small Java application. According to participants' responses, 'Long method' was the top driver for refactoring decision and poor readability along with poor understanding of the code were also among the most important drivers.

3. Study Design

The purpose of the current dissertation is to shed light on the factors that drive developers to resolve TD Items (TDIs) identified in their own code. To achieve this, four PHP open source projects on GitHub were analyzed to obtain commit activity and code debt information. Specifically, Composer, CakePHP, Laravel[18] and Yii2 were included. Composer (composer/composer) is a dependecy manager for PHP, CakePHP (cakephp/cakephp) is a framework to build PHP applications and Laravel

[18] Laravel core (laravel/framework)

(laravel/framework) and Yii2 (yiisoft/yii2) are also well-known PHP frameworks. The criteria for selecting the aforementioned projects are as follows:

- Projects had to be open source and actively maintained up until the time of this dissertation
- Projects had to be widely used by the PHP community: Composer has 5 millions downloads, CakePHP has 2 millions, Laravel has 6 millions and Yii2 has 1 million.
- Projects had to be maintained by many contributors: Composer has 600+ contributors, CakePHP has 500, Laravel has 400 and Yii2 has 800.
- Projects had to be widely recognized by the PHP community: Composer has 11k stars on GitHub, CakePHP has 7k, Laravel has 35k and Yii2 has 11k.

The history of the commit activity was retrieved via the GitHub API and the code base was analyzed by SonarQube[19] to measure TD at every commit snapshot. It should be noted that the commit history includes the last year's commit data of the projects for two main reasons: The aim of the study is to track the most recent developers' activity in order to ask currently active developers to evaluate the importance of the code violations that SonarQube detected. For example, it would not be reasonable to approach a developer that pushed some commits two or three years ago without any recent activity, since he may be currently inactive. The second reason is that the analysis process with SonarQube is costly in terms of time and resources, especially in cases when the TD is measured for every single commit of the project.

As in any similar study, the major challenge was to retrieve sufficient responses as previous experience has shown that people outside the academic community are not always willing to spare time to contribute to academic studies. To increase the likelihood of obtaining a response, the developers have been approached in a way that could potentially attract their interest, as described next.

3.1. Personalized report to participants

Prior to the request for participating in the evaluation of TD items (even just a single one), developers have been provided with a personalized report of their current activity including their commit density, contribution to the overall TD of the project (relatively to the rest of the developers) and the top-five code violations they insert into the code. The report for a random developer (with anonymized information) for project Yii2 is shown in Figure 1. The obtained response rate in this dissertation was 35%.

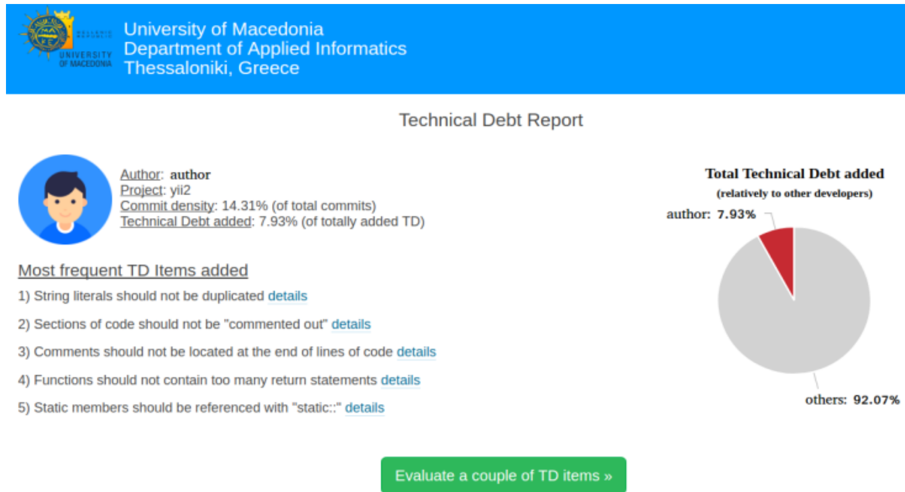


Figure 1. Anonymized TD report of a developer in Yii2

[19] https://www.sonarqube.org/

3.2. Set-up of the Study

At the end of the report each developer was asked to evaluate TD items detected in the project under study. In the evaluation screen the developer was presented with a code violation (assessed TD item) as detected by SonarQube along with some information regarding the violation itself and the file in which the violation was found, so as to provide a spherical view of the TD item before answering. In particular, the evaluator was given the following information regarding the violation (see Figure 2):

- Short description of the TD item
- Suggested solution of the TD item
- Tag categorization (serving as keywords of the TD item)
- Severity of the TD item
- Estimated time to fix the TD item
- The name of the file in which the TD was detected
- The revision of the file
- The code snippet where the TD item was detected
- [20]The change frequency of the file (as percentage)
- [21]The issue fixing frequency of the file (as percentage)
- [22]The total technical debt of the file (as percentage)

Technical Debt Evaluation

⚠ Technical Debt Item: Local variables should not be declared and then immediately returned or thrown
Suggestion: Immediately return this expression instead of assigning it to the temporary variable "$roles". (Line: 415)

Tags: clumsy
Severity: MINOR
Estimated time to fix: 2min

File: framework/rbac/PhpManager.php
File change frequency: 93% (i)
Issue fixing frequency: 98% (i)
Technical Debt of file: 69% (i)
Revision: aa4b49a2e6950596ff8efc1e659708494ec6d253

```
413         $roles = [$roleName => $role];
414
415         $roles += array_filter($this->getRoles(), function (Role $roleItem) use ($result) {
416             return array_key_exists($roleItem->name, $result);
417         });
418
```

"From scale 1 to 5 how important is it to solve the issue above?"

Select | Evaluate

Figure 2. Evaluation screen for a TD item in project Yii2

At the bottom of the screen the developers were asked to evaluate the urgency of the TD item to be fixed in a Likert scale (from 1 to 5), with 1 meaning "*no need to solve it*" and 5 corresponding to "*it is urgent to solve it*". The developers' response to this question served as the dependent variable in the statistical analysis.

[20] This indicates how often the file gets modified, relatively to other files. A percentage of 100% means that the file is the most frequently modified file.

[21] This indicates how often the file gets modified for issue fixing, relatively to other files. A percentage of 100% means that the file produces the most issues.

[22] This indicates the technical debt of the file, relatively to other files. A percentage of 100% means that the file has the highest technical debt.

4. Results and Discussion

4.1. Statistical Analysis

This subsection presents the descriptive statistics on the involved variables and inferential statistics regarding the relationship between the factors that have been considered (explanatory variables) and the agreement of a developer on the resolution of a TDI (dependent variable). Table 1 summarizes the distributions for the categorical variables of the study, whereas Table 2 provides the univariate descriptive statistics of the continuous variables, in which results were expressed as mean (M), standard deviation (SD), median (Mdn), minimum (min) and maximum (max). (Developer Participation indicates whether the participant contributed to the project in which the TD item was found, or not).

Table 1. Frequency distributions for categorical variables

		N	%
Developer Evaluation (Dependent)	Very low	74	27.2
	Low	38	14
	Moderate	60	22.1
	High	49	18
	Very high	51	18.8
Severity	Info	15	5.5
	Minor	58	21.3
	Major	189	69.5
	Critical	10	3.7
Debt characterization	Changeability	14	5.3
	Maintainability	157	59.2
	Reliability	70	26.4
	Security	5	1.9
	Testability	19	7.2
	Missing	7	
Developer Participation	No	105	38.6
	Yes	167	61.5

Table 2. Descriptive statistics for continuous variables

	N	*M*	*SD*	*min*	*max*
Time to fix (in min)	272	10.71	14.28	1	60
TD file (in min)	272	274.63	441.30	2	2828
File modifications ranking	272	84.97	17.47	6	100
File corrections ranking	272	36.18	40.91	0	100

In order to examine the relationship between explanatory variables and outcome responses (Developer Evaluation), the *Generalized Estimation Equations* (GEE) approach is adopted. GEE introduced by Liand and Zeger [12] can be considered as the extension of the Generalized Linear Model, suitable for taking into account the dependence among observations. As in this survey eighteen developers provided their evaluations, each one for one up to eighty-three TD items, there is an imperative need to handle the inherent dependence (or "developer effect"), stemming from the evaluations of the same developers to TD items.

Describing briefly, consider a random sample of observations from n subjects (responses on TD items). Let $\boldsymbol{O}_i^T = \left(O_{i1}, \dots, O_{in_i}\right)^T$ be the column vector of ordinal responses provided by subject $i = \{1, \dots s\}$ where O_{ir} takes values in $\{1, \dots, C\}$. Also let $\boldsymbol{X}_i = (\boldsymbol{X}_{i1}, \dots, \boldsymbol{X}_{in_i})^T$ be a $n_i \times p$ dimensional matrix of repeated p covariates for subject i. Then, the model describing the correlation between the set of covariates and the conditional probabilities of each ordinal response is given by:

$$l[P(O_{ir} \leq c | \boldsymbol{X}_{ir} = \boldsymbol{x}_{ir})] = \beta_{0c} + \boldsymbol{x}_{ir}\boldsymbol{\beta}_1^T \tag{1}$$

For $c = 1, \dots, C-1$, β_{0c} the threshold parameter for level c, $\boldsymbol{\beta}_1$ the row vector of regression coefficients corresponding to covariates and with l a known link function is denoted (*logit* function in this case). The selection of the explanatory variables was based on a backward elimination.

The backward elimination procedure indicated that the covariates *File Modifications Ranking*, $\chi^2(1)$ = 0.030, p = 0.863, *Time to Fix (in minutes)*, $\chi^2(1)$ = 0.512, p = 0.474 and *TD Files (in minutes)*, $\chi^2(1)$ = 1.482, p = 0.223 do not present a statistically significant main effect on responses and for this reason they were dropped out from any further analyses. The final model, after omitting insignificant predictors, indicated that *Severity*, $\chi^2(3)$ = 15.625, p = 0.001, *Debt Characterization*, $\chi^2(4)$ = 12.669, p = 0.013, *Developer Participation (Binary)*, $\chi^2(1)$ = 6.625, p = 0.009 and *File Corrections Ranking*, $\chi^2(1)$ = 3.418, p = 0.064 presented statistically significant main effects on the developer evaluation for TD items.

The parameters of the final model are presented in Table 3, in which the reference categories for factors *Severity*, *Debt Characterization* and *Developer Participation* are "Critical", "Maintainability" and "Yes", respectively. Interpreting the parameter estimates of the model for the factor Severity, the coefficient for the level Info (b = -3.070, SE = 1.374) indicates that the ordered logit for Info TD items, being into a higher evaluation response is -3.070 ($\chi^2(1)$ = 4.990, p = 0.025) less than the reference category (Critical TD items). In other words, the odds for a Critical TD item to be evaluated into a higher category are 21.5 ($1/e^{-3.070}$) times higher compared to an Info TD item.

In addition, the model reveals a statistically significant difference between the odds ratio (*OR*) of Minor and Critical Severity, $\chi^2(1)$ = 7.407, p = 0.006. Regarding Debt Characterization, the findings suggest that Testability debt (b = 1.363, SE = 0.539) is 3.9 times more likely to be evaluated into higher categories compared to Maintainability debt, $\chi^2(1)$ = 6.391, p = 0.011.

In addition, the parameter of the binary predictor Developer Participation, (b = 1.120, SE = 0.430) indicates that TD items presented to developers that have not participated in the project under study at all are almost 3 times more likely to be evaluated to higher categories compared to TDIs presented to developers who contributed to the project. Finally, the coefficient for the covariate File Corrections Ranking, (b = 0.007, SE = 0.004) indicates a marginally significant positive correlation between File Corrections Ranking and Developer Evaluation, $\chi^2(1)$=3.418, p=0.06.

Table 3. Parameters of the final model

Parameter		*b*	*SE*	Hypothesis Test			*OR*	95% OR	
				χ^2	*df*	*p*		*Lower*	*Upper*
Threshold[23]	*Very low*	-2.103	1.154	3.318	1	0.069	0.122	0.013	1.173
	Low	-1.323	1.126	1.381	1	0.240	0.266	0.029	2.419
	Moderate	-0.223	1.044	0.046	1	0.831	0.800	0.103	6.191
	High	0.861	1.053	0.669	1	0.413	2.366	0.301	18.615
Severity:*Info*		-3.070	1.374	4.990	1	0.025	0.046	0.003	0.686
Severity:*Minor*		-2.984	1.096	7.407	1	0.006	0.051	0.006	0.434
Severity:*Major*		-1.409	0.963	2.144	1	0.143	0.244	0.037	1.612
DebtCharacterization:*Changeability*		0.481	0.290	2.742	1	0.098	1.617	0.915	2.857
DebtCharacterization:*Testability*		1.363	0.539	6.391	1	0.011	3.908	1.358	11.241
DebtCharacterization:*Security*		-0.653	1.047	0.389	1	0.533	0.520	0.067	4.049
DebtCharacterization:*Reliability*		0.143	0.266	0.288	1	0.591	1.153	0.685	1.942
Developer Participation:*No*		1.120	0.430	6.783	1	0.009	3.066	1.319	7.123
File Corrections Ranking		0.007	0.004	3.418	1	0.064	1.007	1.000	1.014
Notes: Reference categories Severity:*Critical*, Debt Characterization:*Maintainability*, Developer Participation:*Yes*									

4.2. Discussion of the Results

The distribution of developer responses to the question on whether they agree with the need to resolve a particular TD item are rather uniform, as in 41% of the violations their level of agreement was ‘very low’ or ‘low’, in 22% of the cases their level of agreement was ‘moderate’, while in 37% of the cases they agreed on the need to apply a refactoring for resolving an issue (level of agreement was ‘high’ or ‘very high’).

According to results of the *Generalized Estimation Equations* approach developers appear to be largely influenced by the severity of a TD issue (i.e. Critical, Major, Minor and Info as no Blocking issues were identified). For example, it is 21.5 times more probable that a Critical issue will be classified as needing resolution compared to an Info issue. This finding is reasonable, as the categorization of severity by SonarQube already distinguishes between issues. In other words, it is reasonable that a Critical code issue like “*String literals should not be duplicated*” is perceived as more urgent to be resolved than an Info code issue like “*Comments should not be located at the end of lines of code*”.

The broader characterization of the TD issue also seems to have an effect on the developer’s decision. For example, if an issue pertains to Testability (like “*Expressions should not be too complex*”) it is 3.9 times more probable to be considered as needing resolution than an issue related to Maintainability (like “*Sections of code should not be "commented out*”). Considering that the scanner employed for

[23] In this type of models threshold parameters that define transition points between adjacent categories are estimated for C-1 levels

identifying rule violations in PHP code relied on static analysis, it is reasonable that issues related to Testability, Changeability and Maintainability are considered as more 'real' compared to security/reliability issues which in order to be accurate require further validation by run-time analysis.

Finally, developers do not tend to accept suggestions for revising their own code: it is 3 times more likely that a developer who has not participated in a project agrees with a suggestion to remove a TD issue, than a developer who is a contributor. This might be related to the particular practices within the community of a software project where certain violations are not considered as harmful because the evolution of the project might have been unaffected by their presence.

On the other hand, developers' decisions appear to be unaffected by factors such as the frequency of modifications to the file under study (reflected in the Files Modifications Ranking variable), the time required to fix an issue and the total TD in the examined file. The last two findings could be related to a latent belief that automated quality analysis tends to overestimate the magnitude of problems and thus these factors might be subconsciously overlooked. The frequency by which a file undergoes modification, under normal circumstances, should be driving factor; for example, for a file that has never been the subject of maintenance there is probably limited urgency to resolve its TD issues. However, it appears that developers tend to focus on the problem per se, rather than the surrounding context. Of course, a relevant threat is related to whether the respondents really understood the concept of the presented variables. These findings can be valuable to researchers and practitioners by guiding the design of more efficient tools that suggest refactorings with a higher probability of being adopted by the developers.

5. Threats to Validity

In this section major threats to the validity of the present study are listed. With regard to statistical conclusion validity it should be stressed that the small sample size unavoidably affects the conclusions regarding the extent of the observed relationships between the explanatory and output variables. Further investigation by collecting a larger set of responses is required to increase the confidence in the identified relationships. With regard to the construct validity of the study, it should be acknowledged that despite the effort to facilitate the response of the participants, by offering an easy-to-use web application, it is not certain that they have correctly interpreted the presented pieces of information around the examined code fragment and TD issues. Finally, the conclusions should be cautiously generalized to other projects, languages, development models and proprietary software as this kind of studies are subject to external validity threats.

6. Conclusions

Existing software quality tools can yield extremely long lists of refactoring suggestions, deterring developers from adopting them. Thus, there is a need to determine which refactoring opportunities make sense for the developers depending on their background, nature and importance of the problem, surrounding code context, etc. This chapter presents the results from an ongoing study on various factors that potentially drive open-source software developers to accept or reject a suggestion to resolve a TD item.

References

[1] A. Hunt and D. Thomas, *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional, 1999.

[2] T. Amanatidis, A. Chatzigeorgiou, A. Ampatzoglou, and I. Stamelos, "Who is Producing More Technical Debt?: A Personalized Assessment of TD Principal," *Proceedings of the XP2017 Scientific Workshops*, USA, 2017, p. 4:1–4:8.

[3] D. Silva, N. Tsantalis, and M. T. Valente, "Why We Refactor? Confessions of GitHub Contributors," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, New York, NY, USA, 2016, pp. 858–870.

[4] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, "Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, USA, 2015, pp. 50–60.

[5] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, and A. D. Lucia, "Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 101–110.

[6] A. Potdar and E. Shihab, "An Exploratory Study on Self-Admitted Technical Debt," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 91–100.

[7] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?," in *35th International Conference on Software Engineering (ICSE)*, 2013, pp. 672–681.

[8] R. O. Spínola, A. Vetrò, N. Zazworka, C. Seaman, and F. Shull, "Investigating technical debt folklore: Shedding some light on technical debt opinion," in *4th International Workshop on Managing Technical Debt (MTD)*, 2013, pp. 1–7.

[9] M. Kim, T. Zimmermann, and N. Nagappan, "A Field Study of Refactoring Challenges and Benefits," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, New York, NY, USA, 2012, p. 50:1–50:11.

[10] Y. Wang, "What motivate software engineers to refactor source code? evidences from professional developers," *IEEE International Conference on Software Maintenance*, 2009, pp. 413–416.

[11] M. V. Mäntylä and C. Lassenius, "Drivers for Software Refactoring Decisions," in *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, New York, NY, USA, 2006, pp. 297–306.

[12] K.-Y. Liang and S. L. Zeger, "Longitudinal data analysis using generalized linear models," *Biometrika*, vol. 73, no. 1, pp. 13–22, Apr. 1986.

Chapter VII. BENCHMARK OF TECHNICAL DEBT LIABILITIES

The work of this chapter was submitted for publication (and revised) to the Empirical Software Engineering Journal (EMSE)

Chapter Summary

Software teams are often asked to deliver new features within strict deadlines leading developers to deliberately or inadvertently serve "not quite right code" compromising software quality and maintainability. This non-ideal state of software is efficiently captured by the Technical Debt (TD) metaphor, which reflects the addition-al effort that has to be spent to maintain software. Although several tools are available for assessing TD, each tool essentially checks software against a particular ruleset. The use of different rulesets can often be beneficial as it leads to the identification of a wider set of problems; however, for the common usage scenario where developers or researchers rely on a single tool, diverse estimates of TD and the identification of different mitigation actions limits the credibility and applicability of the findings. The objective of this chapter is two-fold: First, to evaluate the degree of agreement among leading TD assessment tools. Second, to propose a framework to capture the diversity of the examined tools with the aim of identifying few "reference assessments" (or class/file profiles) representing characteristic cases of classes/files with respect to their level of TD. By extracting sets of classes/files exhibiting similarity to a selected profile (e.g., that of high TD levels in all employed tools), a basis can be established that can be used either for prioritization of maintenance activities or for training more sophisticated TD identification techniques. The proposed framework is illustrated through a case study on fifty (50) open source projects and two programming languages (Java and JavaScript) employing three leading TD tools.

1. Introduction

Throughout the software lifecycle, practitioners speed up the development process by compromising software quality and maintainability in favor of shorter time-to-market. This compromise has been effectively captured by the concept of ***Technical Debt*** (TD), as coined by Ward Cunningham [1], offering an analogy to the financial debt. In financial debt, one party borrows capital from another party and repays it back with some added interest. In the TD metaphor, the development team 'borrows' a certain amount of effort by delivering non-ideal code and repays it gradually in future iterations in the form of additional time and effort to perform maintenance on the non-ideal code. The increased maintenance effort, which is caused by the degradation of software maintainability, is considered as the "interest" that the development team has to pay in the long term. In contrast to financial debt, TD is hard or even impossible to measure accurately. The suggested practice, according to the OMG specification on Automated Technical Debt Measure (ATDM)[24], is to consider as ***principal*** of TD (at the source code level) the total effort required to eliminate TD items, which are inefficiencies that have been identified in a software artifact under an established ruleset. However, even if developers are aware of parts of the code that "*do not feel right*" it is challenging to associate an exact numerical estimate with every rule violation. Software modules evolve over time and subtle or major changes in their TD might be incurred by the transition from one commit to the next, rendering the accurate monitoring of TD even more demanding.

[24] https://www.omg.org/spec/ATDM/About-ATDM

The limitations on accurately measuring TD lead to various shortcomings in both academia and industry, in the sense that one cannot control (or manage) what he/she cannot measure [2]. Despite the fact that several tools are available for measuring and monitoring TD (notable examples include CAST AIP[25], Squore[26], and SonarQube[27]), either commercial or open-source ones, the community has not concluded on a state-of-the-art solution that could be used as a basis for measuring TD (a full list with TD measurement tools that were found during current research is presented in Section 2). Some shortcomings whose roots lie in the lack of a well-established way for assessing (i.e., measuring and identifying) TD principal, are presented in Figure 1.

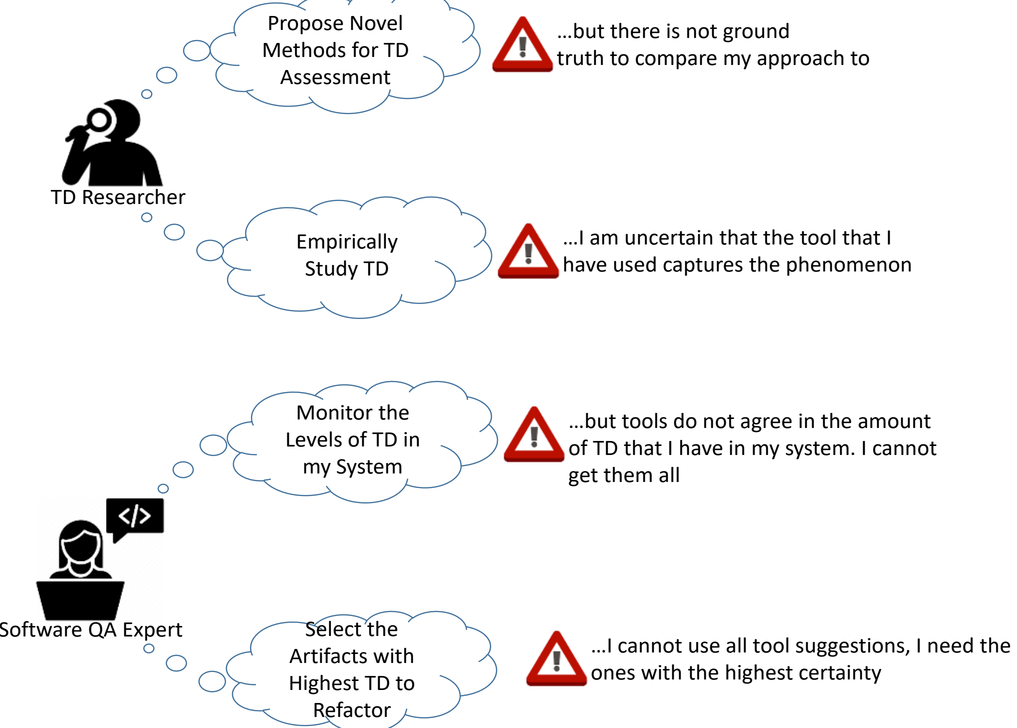


Figure 1. Shortcomings from diverse TD measurements

Shortcomings in Research: The lack of a ground truth, even a commercial one, leads to construct validity threats in almost any kind of quantitative empirical study in the field, in the sense that it is not certain that any metric that attempts to capture TD principal is accurately measuring the real-world phenomenon. This problem does not lie only on limitations of the tools per se, but also on the underlying methodologies. In particular, each tool follows its own approach for detecting and measuring TD based on its own ruleset, while another tool might be based on an entirely different ruleset yielding a different amount for the total TD, but also pointing to different parts of the code that need to be mitigated. Moreover, there are several research efforts trying to associate TD items (i.e., violations of coding practices in software artifacts, which according to the OMG Specification on ATDM – see footnote in first page, are considered instances of TD principal) with quality attributes of software. For example, studies have focused on the relation between TD principal and the presence of crosscutting concerns in software requirements [3], the existence of modularity violations, code smells and static analysis issues [4], code size, duplication and complexity [5] and architecture flaws [6]. However, every such approach is heavily dependent on the employed tool for suggesting the ground truth, that is, the modules that

[25] https://www.castsoftware.com/
[26] https://www.vector.com/int/en/products/products-a-z/software/squore/squore-software-analytics-for-project-monitoring/
[27] https://www.sonarqube.org

actually have TD liabilities and need to be fixed. Obviously, if each tool identifies high-TD modules in a different way, the generalizability of these approaches is threatened to a large extent.

Shortcomings in Practice. Despite the widespread adoption of the TD metaphor, it is far from clear which tool IT managers should trust for monitoring TD, or deciding the mitigation actions to be applied. One option would be to employ more than one TD tools for the evaluation of their software, but this is a costly one, since most of the existing tools are available only with a commercial license. Moreover, someone should also consider the effort to deploy the tools on their premises, configure them properly and eventually familiarize development tools with their usage. In addition to that, even with the use of multiple tools, the union of all possible fixes suggested by different tools would yield an unrealistic amount of suggestions which would end up (even if they were accurate enough) to be useless in practice.

Acknowledging the widespread adoption of the TD metaphor and the inherent limitation of existing tools to capture TD principal in a globally accepted way that best fits developers' needs [7], this part of dissertation aims at: (a) systematically ***investigating the degree of agreement*** among state-of-the-art TD measurement tools on identifying and prioritizing TD principal (i.e. the effort to remediate inefficiencies) at class/file level; and (b) proposing an ***agreement-based benchmark approach*** that contributes to: (i) the exploration of all *feasible assessments of TD principal* provided by a set of alternative TD tools, (ii) the *identification and characterization of few divergent "reference assessments"* (or archetypes); and (iii) the *extraction of a subset of modules* for which all employed tools agree on the presence of a high amount of TD principal, thus serving as an agreement-based benchmark of the "*validated*" top-rated classes/files[28] in terms of TD principal assessment.

To achieve the former goal, a well-known inter-rater agreement coefficient is employed, namely the *Kendall's W coefficient of concordance* [8]. Regarding the second goal (benchmarking process), the *Archetypal Analysis* (AA) [9] is adopted, which is a multivariate statistical methodology that explores a multidimensional space of measurements with the aim of identifying a set of few reference points, namely the archetypes, located on the boundaries of the swarm of given points. The derived archetypes (or reference points) represent divergent profiles in the examined space, whereas the methodology encompasses a mechanism for the evaluation of resemblance coefficients contributing to the evaluation of similarity for each point to the derived archetypes.

To this end, three well-known tools that measure TD have been employed to analyze 50 open source projects (25 Java and 25 JavaScript projects). The results of the proposed methodology are automatically reported through a web-based interactive toolbox to facilitate researchers and software practitioners to reproduce and explore the findings of the current work and easily retrieve a suitable benchmark for further experimentation (e.g., the training of other statistical or machine learning approaches to identify TD items). The ***TD Benchmarker*** toolbox is implemented using the Shiny framework[29] taking advantage of the R statistical language[30] in an easy-to-use frontend. The toolbox is a free and academic on-going research project developed by Statistics and Information Systems Group (STAINS)[31] at Aristotle University of Thessaloniki, Greece and is accessible through the paper's web page[32], at the website of the Software Engineering Group of University of Macedonia[33], Greece.

Apart from the empirical results and the extension of the body of knowledge in the field of TD management, an actionable outcome of this dissertation is the provision of an agreement-based benchmark set of the most high-TD classes as indicated by the three tools altogether. The agreement-

---

[28] The term 'class' refers to the unit of analysis for Java projects, while the term 'file' refers to the unit of analysis for JavaScript projects. Throughout the paper the term 'class' is primarily used for simplicity, but both units of analysis are considered, accordingly.

[29] https://shiny.rstudio.com

[30] https://www.r-project.org

[31] http://stains.csd.auth.gr

[32] https://se.uom.gr/index.php/projects/technical-debt-benchmarking

[33] https://se.uom.gr

based benchmark is expected to alleviate the aforementioned limitations either directly or indirectly: regarding researchers the benchmark can be exploited for methodologies aiming at identifying TD items (targeting either high recall, or high precision), whereas it is also expected to aid practitioners since it will contribute to the development of novel tools that will be able to predict these items. More details on the implications of the benchmark for researchers and practitioners are provided in Section 5.

The rest of the chapter is organized as follows: Section 2 presents available TD assessment tools that have been located throughout the current research, while it is also explained why the three TD tools were ultimately used. In Section 3, the case study design is presented along with the objectives, the research questions, the data analysis and the methodology. Section 4 presents and discusses results and in Section 5 the implications to researchers and practitioners are highlighted. Section 6 unfolds possible threats to validity while in Section 7 related work of previous studies on comparison of TD tools and benchmarks in software maintenance is provided. Finally, conclusions are discussed in Section 8.

2. TD Assessment Tools

This section discusses TD assessment tools that either have been proposed in the context of research efforts (usually open-source or free) or are available as commercial software, by providing a brief description of their capabilities. Then more details on the three tools that have been selected for this dissertation are provided, explaining the rationale for their selection.

During the previous years, numerous TD assessment tools have emerged; these tools are able to measure TD either in terms of cost or effort/time to repay TD. To identify as many tools as possible, a non-systematic literature search has been conducted, including grey literature (such as websites):

- Literature search: Regarding literature search, it relied on the IEEE Xplore[34] and ACM Digital Library[35] search engines. The search string was applied on the title and abstract fields and had the following form: “technical debt” AND (measurement OR assessment OR estimation) AND (tool OR platform). The studies that have been returned from the aforementioned search were gathered and those which neither introduce nor mention any TD tool in their title or abstract were filtered out.
- Web search: Throughout web search, major search engines such as Google, Bing and Yahoo were used, using the same query. The results led to either the landing pages of the websites of the companies that own the tools or articles introducing most well-known tools for assessing TD.

Right below follows a short description of the TD assessment tools that have been located throughout the current research. For each tool the study and the year are provided in which it was first introduced or presented. The actual versions of the employed tools at the time of this dissertation are provided in the end of this section.

AnaConDebt [10] is a tool that focuses on Architectural Debt. Since a change in the architecture of a project can be really expensive and time consuming it is important to decide if and when this change should be implemented. The tool uses a large list of internal and external factors to estimate more accurately the future principal and interest. It helps managers to decide when it is the right time to refactor the code of their software.

CAST AIP [11] contains several sub-tools in order to provide the entire quality profile for the project. Health dashboard, Engineering dashboard, Security dashboard, CAST Appmark which is a

[34] https://ieeexplore.ieee.org

[35] https://dl.acm.org/

benchmarking base to use as a comparison standard and CAST Enlighten with Imaging system that offers a visualization of the project. This tool helps companies to perform "Shift Left" techniques to detect the issues of a project in early stages of its life cycle. This way the cost of fixing the issues is more tolerable. The tool implements the C-CPP, CISQ, CWE, NIST-SP-800-53R4, OMG-ASCQM, OWASP, PCI-DSS-V3.2.1 and STIG-V4R8 standards. By performing static analysis, a list of issues is created. Only a part of the problems will be solved and this part defines the technical debt metric.

CodeScene [12] serves as a mean to preserve the quality of the code of the automated tests. It combines repository mining with static code analysis and machine learning. Static analysis can detect the problems in the project, but since the source code is treated as of the same importance, repository mining is necessary to recognize behavioral data and social factors that can affect future decisions of refactoring. The results of the metrics may have different meaning depending on the characteristics of each project. Machine learning is used to identify patterns in order to prioritize these metrics and assign them the appropriate weight. The final result of the tool is a catalogue with the problematic files ranked by their total impact.

DebtFlag [13] is a tool for capturing, tracking and resolving technical debt in Java systems. It consists of two parts; one plug in for Eclipse IDE which is responsible to collect the data from the source code, and one web application to visualize the results. These two applications connect via a database. The collected data is structured using the TDMF form, which was extended to cover the tool's needs. The tool offers the results in such a way that can be used to manage technical debt in two levels; project level and implementation level with micromanagement.

Debtgrep [14] is an inhouse tool developed by Ericsson 4G 5G Baseband and its purpose is to prevent technical debt. It uses a file where all rules are declared using regex. The rules can contain forbidden words to restrict the usage of API and deprecated methods and also guidelines for design and architectural rules. The rules can be applied only to a specific part of code such as new code. This tool supports the communication between the developing team members and enhance the consistency and the uniformity of the project.

DV8 [6] is a commercial extension of Titan [15]. DV8 functions with DRSpaces [16], which are groups of system's files that are architecturally related. Within DRSpaces, DV8 computes three modularity metrics (Decoupling level, Propagation Cost and Independence Level) and detects six architecture anti-patterns (Clique, Package Cycle, Improper Inheritance, Unstable Interface, Crossing and Modularity Violation). DRSpaces (i.e. the subsets of architecturally related files) that are involved in a selected set of issues are called 'architecture roots'. The tool calculates the added maintenance cost due to each instance of each anti-pattern, and the added maintenance cost of each architecture root. The source code analysis is performed by the Understand tool[36].

Kiuwan[37] is a proprietary code analysis tool that supports numerous programming languages and is capable of integrating with several IDEs. It can be obtained under a commercial license and it can also be tested within a free trial period.

NDepend [17] is a static analysis tool for .NET projects available in Visual Studio Market Place. It offers a variety of code quality metrics and a visualization of the dependencies in the project. The tool handles the source code as a form of database, and the user can define new evaluation rules using LINQ to perform queries on it. Other features of the tool include reporting service and the ability of comparison between the generations of the same project.

SonarQube [18] is a widely known tool used to track the quality and maintainability of source code. The tool implements the MISRA, CWE, SANS and CERT rule standards to provide measurements

[36] https://scitools.com/
[37] https://www.kiuwan.com/

regarding complexity, duplications, code issues, maintainability, quality gates in combination with technical debt, reliability, security, project size and test coverage. In addition, there are many plugins to extend the available utilities, such as WebDriver for Selenium test analysis or AEM Rules set for Adobe. The measurement of technical debt is an important component of SonarQube. The tool calculates the debt by multiplying the number issues of each type with the average time the specific issue type needs to be fixed. Then the time is multiplied with the cost for each man-day. The average time and the cost can be configured by the user. It uses the SQALE method and provides a technical debt pyramid to help making decisions prioritizing tasks.

Squore [19] consists of three smaller tools. The first one, the analyzer, is used to collect data from different sources (source code, tests and hardware component information) and build the project's hierarchy tree. Then a more detailed measurement takes place for each one of the nodes based on the ISO, HIS, SPICE and MISRA rule standards. Last but not least, the tool also offers a dashboard for the visualization of the results. The tool can be a part of Jenkins continuous integration and can also recognize which files are most important to have Unit Tests in order to improve the efficiency.

TD-Tracker [20] is a web application, which provides a structured way to create a catalogue with the issues in a project. The protocol, which is implemented, consists of three stages. For the first stage there is a data collector where the problems are identified and a list is populated. The input data can come from either an external source where, with appropriate mapping, the data can be stored directly to the database of the application, or the integration with GitHub. After finishing the collection, the second stage begins where a semi-automated task takes place. A user has to review the previous list with the issues, and decide which of them are actual problems that need to be solved. Then there is the third stage with the longest duration of all three. In this stage a user assigns tasks related to technical debt and also monitors the progress of them.

TEDMA [21] is an open tool, which analyzes different indices related to technical debt during the evolution of a project. It is open to integrate with third party tools to extend the analysis. It consists of three layers. The first is called Data Layer and holds the processes used to gather information about the project, which is examined. Currently, Git repositories are used as data input. The second is the Service Layer where there are three basic services. (i) Data loader service is responsible for offering the source code in a processable form to the tool. Then analyzers such as PMD and Findbugs detect code smells and problems. (ii) Statistics service uses R to perform statistical analysis of the data. The analysis is performed at file level but it can be extended to other levels of abstraction. (iii) Technical debt management model service uses models in Java and R to support decision-making. The last layer is the Presentation Layer which is responsible for documentation and visualization.

VisminerTD [22] is an open source web tool which monitors and manages technical debt comparing the results between different project's versions. When an issue is detected it can be tracked to determine whether its TD was paid off or not. It uses the Repository Miner tool to collect data and metrics from code repositories. VisminerTD uses queries to the database of the Repository Miner to gather the preferred information and present them to the user via a friendly interface. A set of graphical views are available to setup the search settings and then manage the technical debt items.

Table 1 lists the tools that have been identified along with information, such as the website with contact or download information, the corresponding study in which it was first introduced or presented, the type of license under which the tool is available (commercial/free), the programming languages that the tool supports for static code analysis and the type(s) of TD that it captures (as identified in previous studies [23], [24]). TD types refer to specific categories of TD (e.g., architectural, design, code) or sub-categories based on the cause of TD (e.g., architectural TD can be caused by architecture smells) [24].

Table 1. List of identified TD assessment tools

TD Tool (Website)	Study	License	Supported Programming Languages	Captured TD Type(s)
AnaConDebt (https://anacondebt.com/node/7)	(Martini and Bosch, 2016) [5]	commercial	Java	Architectural
CAST AIP[38] (http://www.castsoftware.com/)	(Curtis et al., 2012) [6]	commercial	Java, ASP, C/C++, Android, IOS, .NET, PHP, Python, ABAP, SQL (and more, see full list at website)	Architectural, Code, Defect
CodeScene (https://codescene.io/)	(Tornhill, 2018) [7]	commercial	C/C++, C#, Java, JavaScript, TypeScript, Python, Go, Visual Basic .Net, PHP, Ruby (and more, see full list at website)	Code, Design
DebtFlag (-)	(Holvitie and Leppänen, 2013) [8]	-	Java	Code
Debtgrep (-)	(Arvedahl, 2018) [9]	Inhouse use only	Language agnostic	Architectural, Code, Design, People
DV8 (https://archdia.com/pages/dv8-user-guide) Understand: third party tool for source code analysis (https://scitools.com/)	(Nayebi et al.) [6]	commercial	Java, JavaScript, C/C++, C#, Python , PHP and more (see full list here: https://scitools.com/feature/supported-languages/)	Architectural
Kiuwan (https://www.kiuwan.com/)	-	commercial	ASP.NET, C, C#, C++, Java, JavaScript, JSP, PHP, Python, VB.NET, SQL, Ruby (and more, see full list at website)	Code
NDepend (https://www.ndepend.com/)	(Chopra and Sachdeva, 2015) [10]	commercial	.NET	Architectural, Code, Design, Test
SonarQube (https://www.sonarqube.org/)	(Campbell and Papapetrou, 2013) [11]	free	C/C++, C#, CSS, Go, Java, JavaScript, PHP, Python, Ruby, TypeScript, VB.NET (and more, see full list at website)	Architectural, Code, Design, Defect, Test
Squore (https://www.squoring.com/en/produits/squore-software-analytics/)	(Baldassari, 2013) [12]	commercial	Ada, C, C++, C#, Java, Cobol, PL, SQL, ABAP, PHP, Python	Code, Test

[38] We will refer to it as “CAST” from this point on

TD Tool (Website)	Study	License	Supported Programming Languages	Captured TD Type(s)
TD-Tracker (http://www2.fct.unesp.br/grupos/lapesa/tdr/)	(Foganholi et al., 2015) [13]	free	Java, JavaScript, PLSQL, Apache Velocity, XML, XSL	Code, Design, Defect, Documentation, Infrastructure, Test
TEDMA (-)	(Fernández-Sánchez et al., 2017) [14]	-	Java	Architectural, Code
VisminerTD (https://visminer.github.io/)	(Mendes et al., 2019) [15]	free	Java	Architectural, Build, Code, Design, Defect, Documentation, Requirement, People, Test

Employed TD Assessment tools. Despite the goal to include in the study as many tools as possible, it has not been possible to employ all of the above tools for the measurement of TD for the target systems. Each tool had to fulfill the following conditions in order to be included in the study. Table 2 presents which tools have been included in the study and which have been excluded (failing to satisfy all of the following conditions).

- Condition 1: The tool had to be accessible somehow (download link, ftp server, etc.) with comprehensive and sufficient documentation.
- Condition 2: The tool had to be able to analyze Java and JavaScript code (as the target systems of the study are open source Java and JavaScript projects).
- Condition 3: It was necessary to be able to obtain academic or research license for commercial or proprietary tools. For non-proprietary tools the condition was considered fulfilled.
- Condition 4: The tool had to provide an aggregate TD Principal index at class/file level, expressing effort in time or monetary terms, to remediate the identified inefficiencies (OMG Specification on ATDM[39]). Estimation of TD only at project level cannot be exploited to extract a benchmark set of most high-TD classes (for Java projects) and files (for JavaScript projects). This criterion is important for guaranteeing the uniformity of tools' output, so that the results are comparable

[39] https://www.omg.org/spec/ATDM/About-ATDM

Table 2. List of TD tools with the conditions that they satisfied for their inclusion

TD Tool	Condition 1	Condition 2	Condition3	Condition 4	Tool used?
AnaConDebt	✓	X	X		no
CAST	✓	✓	✓	✓	**yes**
CodeScene	✓	✓	✓	X	no
DebtFlag	X	X	✓		no
Debtgrep	X	✓	X		no
DV8	✓	✓	✓	X	no
Kiuwan	✓	✓	X		no
NDepend	✓	X	✓		no
SonarQube	✓	✓	✓	✓	**yes**
Squore	✓	✓	✓	✓	**yes**
TD-Tracker	✓	✓	✓	*could not deploy*	no[40]
TEDMA	X	X	✓		no
VisminerTD	✓	X	✓		no

*In case a tool did not fulfill Conditions 1 - 3 or could not be successfully installed and deployed, Condition 4 could not be checked and thus the field was left blank.

Ultimately, three tools were included in the study, namely CAST (version 8.3, year 2018), Squore (version 19.0, year 2019), and SonarQube (version 7.9, year 2019). All three tools are major TD tools, widely adopted by software industries and researchers and actively maintained, including comprehensive documentation.

3. Case Study Design

3.1. Goal and Research Questions

The goal of this chapter described according to the Goal-Question-Metric (GQM) approach [25], is as follows: "**analyze** the TD of software projects **for the purpose of** assessing the level of agreement of state-of-the-practice TD assessors (tools) and forming agreement-based TD benchmarks of high-TD (or low-TD) classes **with respect to** the estimated level of principal, **from the point of view of** software researchers and practitioners **in the context of** Technical Debt Management (TDM)". For the sake of generalization, the assessment of the level of agreement among tools was performed for two programming languages, namely Java and JavaScript. The analysis of the two populations enables a meta-analysis in which it can explored if the use of a different language has an effect on the level of agreement. The exploration of the programming language as a factor affecting the level of agreement between tools is performed for each one of the following research questions:

RQ1: To what extent do the assessors (tools) agree in the ranking of classes in terms of TD measurement?

RQ1 aims at investigating the degree to which widely employed TD tools agree upon the identification and assessment of TD at class level. The investigation of this RQ provides an insight to the diversity of the rules examined by each tool, in the sense that a low level of agreement essentially means that tools check for different rule violations. With a non-satisfying degree of agreement, it would be pointless to proceed with the benchmarking process and seek classes, which are identified as equally high-TD (or low-TD) by all assessors. Thus, RQ1 serves as a gate for the rest of the study.

[40] TD-Tracker was not included because it was not possible to install and deploy it successfully.

RQ2: How many archetypes (reference assessments) are required to capture the diversity of the tools?

The TD of classes in any examined system, as measured by the employed tools, form a set of observations in a multidimensional space, in which each dimension represents TD evaluations provided by a specific tool. RQ2 aims at exploring this multidimensional space and determine the optimal number of archetypes, located on the boundaries of this space, so as to efficiently capture the diversity of all feasible assessments provided by the set of the examined TD tools. For example, this RQ can answer, whether few reference assessments are able to approximate the convex hull of the TD evaluations, which practically means low diversity among TD assessors or whether a higher number of archetypes would be required to accurately characterize the spectrum of TD measurements for a given system.

RQ3: Which are the characteristics of the extracted archetypes?

RQ3 aims at characterizing the extremal points that accurately encompass the space of TD measurements for all examined classes. The identified reference assessments essentially form a set of distinct archetypes, i.e., class profiles according to the measured level of TD. Two expected archetypes correspond to the profiles of classes having high or low TD based on the results of all employed tools. However, other archetypes may be identified based on the shape of the space of the obtained TD measurements.

RQ4: Which classes should be selected to form an agreement-based benchmark of top-TD modules?

To facilitate the work of developers or researchers who seek a golden set of classes that can be safely assumed to be high-TD or low-TD, this RQ aims at formally extracting sets of classes which are close to a selected class profile or archetype. Retrieving for example the classes, which are in the close vicinity of the archetype depicting high TD in all employed tools, a development team can be confident that these classes suffer significantly from rule violations. Similarly, a researcher can use such a benchmark for training effective machine learning techniques to identify TD based on different parameters of the code, people or processes involved in the development.

3.2. Selection of Cases

For this case study fifty (50) open source projects were analyzed (listed in Table 3). The selected projects, which are 25 Java and 25 JavaScript projects, have been analyzed considering their classes (for Java) and their files (for JavaScript) as units of analysis. The choice of classes/files as units of analysis allows us to trace the existence of TD at a low level of granularity, providing a common ground for comparison among the three tools. The criteria for selecting the 50 projects were the following:

- All cases had to be Java and JavaScript projects stored on public repositories.
- All selected cases had to be among the most popular repositories, with more than 3K stars in GitHub.
- In order to obtain a representative dataset, the selected projects had to vary in terms of size, per language.
- All cases had to be actively maintained till the time of this dissertation. This was not a strict criterion since projects with a release around the last year before the project selection process were not excluded from the study.

3.3. Data Collection

The source code (excluding test files) of each project was analyzed three times: one time for each of the employed TD tools. All three tools provide a metric of the total effort needed to eliminate technical debt in each class/file. This is the metric that was chosen for analysis since it provides a common ground for comparison. An issue that had to be addressed was that each tool has a different way to provide the results of its analysis. It was necessary to convert the result sets from each tool to the same form so as to proceed with further data processing.

- SonarQube has a WEB API available, so with the use of appropriate tools the results have been gathered in json format. The API allows the filtering of the results in order to exclude test and properties files. SonarQube provides the results grouped by file. Besides file name, the number of the issues for each severity level, blocker, critical, major, minor and info, was summed up to the total amount of issues of each class. All of them contribute to the SQALE index of the file, which is the metric depicting the effort to eliminate TD.
- Squore provided the results in .csv files, which could be exported through platform's user interface. In this case a parser was necessary to read the .csv files. Using the previous SonarQube exports as reference, the files were filtered to exclude test and property files as before. Blocker, critical, major and minor issues were summed up to get the total issues for each class. Technical debt metric is provided in man days and man hours and it had to be converted in minutes to form a canonical technical debt index with the same units as for the previous tool.
- CAST provides metrics for the total project and not per file through its user interface. In this case, the results were retrieved directly from the database schema that the software uses during the code quality analysis. With appropriate SQL query, which was provided by the CAST team, csv files were extracted containing a list of total occurrences of each issue per class. With a new parser these issues were grouped, aggregating the TD in minutes and the total violations per class. Then again, the files of the classes were filtered with those of SonarQube as reference (test and property *files* were filtered out).

To obtain a common and structured form of the results, the exports from the tools were transformed into XML files. As a result, an XML file per project for every tool was generated. The XML contains all the classes/files with some TD in the system, along with the total issues detected in the class and the amount of TD as calculated by the corresponding tool. With the results in the same form it was possible to merge them into a single dataset. This dataset was finally grouped by class for Java and by file for JavaScript projects, containing the path of the file and the TD of the class/file as calculated by each tool. The dataset for the 25 Java and the 25 JavaScript projects can be found in the paper's web page[41].

[41] https://se.uom.gr/index.php/projects/technical-debt-benchmarking

Table 3. Characteristics of analyzed projects

Java				JavaScript			
Project	**Description**	**LOC**	**Version**	**Project**	**Description**	**LOC**	**Version**
arduino	Physical computing platform	27K	1.8.10	ace	Code editor	117K	1.4.8
arthas	Java Diagnostic tool to troubleshood production issues	28K	3.1.7	angular.js	Web development framework	53K	1.7.9
azkaban	Workflow manager	79K	3.81.0	atom	Text editor	138K	1.44.0
cayenne	Java object to relational mapping framework	348K	3.1.2	bluebird	Promise library	20K	3.7.2
deltaspike	CDI management	146K	1.8.2	bower	Front end package management	10K	1.8.8
exoplayer	Android media player	155K	2.11.1	brackets	Code editor	129K	1.14.1
fop	Print formatter using XSL objects	292K	2.3	Chart.js	Chart designer	10K	2.9.3
gson	Java library to convert Java Objects to JSON	25K	2.8.6	exceljs	Excel Workbook Manager	23K	3.8.0
javacv	Wrappers of commonly used libraries	23K	1.5.2	fabric.js	Framework for HTML5 canvas element	20K	4.0.0
jclouds	Toolkit for java cloud applications	482K	2.0.2	jquery	Javascript library	20K	3.4.1
joda-time	Date and time handling	86K	2.10.5	karma	Tool for test driven development	5K	4.4.1
libgdx	Game development framework	280K	1.9.10	Leaflet	Mobile friendly interactive maps	24K	1.6.0
maven	Software project management and comprehension tool	106K	3.5.4	less.js	Language extension for CSS	12K	3.11.1
mina	Network application framework	35K	2.0.19	moment	Parsing validating manipulating and formatting dates	183K	2.24.0
nacos	Cloud application and microservices build and management	60K	1.1.4	mongoose	Tool for MongoDB object modeling	22K	5.8.12
opennlp	Natural Language Processing toolkit	93K	1.8.4	mysql	MySQL protocol implementation	8K	2.18.1
openrefine	Data management	69K	3.2	node	Node.js JavaScript runtime	130K	13.9.0
pdfbox	Library of processing pdf documents	213K	2.0.9	pdf.js	PDF viewer	69K	2.2.228
redisson	Java Redis client and Netty framework	133K	3.12.0	plotly.js	Chart design library	92K	1.52.2
RxJava	Composing asynchronous and event-based programs with observable sequences	310K	3.0.0	pm2	Production process manager	15K	4.2.3
testng	Testing framework	85K	7.1.1	prettier	Code formatter	25K	1.19.1
vassonic	Performance framework for mobile websites	7K	3.1.1	sails	Realtime MVC Framework for Node.js	10K	1.2.2
wss4j	Java implementation for security standards in web applications	136K	2.2.2	sequelize	Node.js ORM	17K	5.21.4
xxl-job	Distributed task scheduling framework	9K	2.1.2	webpack	Bundler for js files for usage in a browser	36K	4.41.6
zaproxy	Security tool	187K	2.9.0	yarn	Dependency management	24K	1.22.0

3.4. Data Analysis Methodology

This section presents background information necessary for facilitating the understanding of the statistical methodologies used to address the research questions of the current part of dissertation.

3.4.1. Inter-rater Agreement (RQ_1)

For the formal representation of the experimental setup, consider that the collection of TD assessments generated by all three tools, as described in Section 3.3, resulted in a $n \times p$ matrix (Table 4), in which, each row represents a class, whereas each of the p column vectors provides the rankings of TD measurements evaluated by a specific tool for a given class. At this point, it should be clarified that in the proposed approach the rankings instead of the raw TD measurements have been utilized, since the intention was to keep the dataset immune to variations of TD measurements due to different scales among the three tools. Indeed, a tool might follow a stricter ruleset for the measurement of TD which might result in much higher TD of classes compared to the assessments of the rest of the tools. However, the ranking of the measurements among all tools remain unaffected by absolute values and thus is a more suitable approach for comparison. As far as the ranking mechanism concerns, the fractional ranking approach was adopted, in which the sample ranks of the values in a vector are computed, whereas in cases of ties the average of the ordinal rank (or fractional rank) is assigned to each tied observation.

Table 4. Representation of the dataset from the TD assessment results from each employed tool[42]

Class	Tool 1	Tool 2	…	Tool p
C_1	r_{11}	r_{12}	$\cdots$	r_{1p}
C_2	r_{21}	r_{22}	$\cdots$	r_{2p}
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
C_n	r_{n1}	r_{n2}	$\cdots$	r_{np}

For reasons of simplicity, the methodology of the proposed framework is presented on a demonstrative example (opennlp project) utilizing the TD assessments from two tools (CAST and Squore). In this case, the TD assessments can be visualized through a scatter plot (Figure 2), in which each point represents a specific class with coordinates the TD rankings evaluated by the CAST (x-axis) and Squore (y-axis) tools.

The exploration of the pattern for the swarm of points provides certain information regarding the agreement of the employed tools. More precisely, it seems that there is a subset of classes lying to the upper right corner that are identified as the most high-TD (high rankings for TD measurements) by both tools. On the other hand, the inspection of the graph indicates also that Squore identifies a subset of classes that accumulate the lowest TD assessments but at the same time, these specific classes present an amount of TD ranging from the lowest up to the highest ranks according to the CAST tool. Finally, there is also a small number of classes assessed as high-TD by the Squore tool, but at the same time, the CAST tool tags them as classes accumulating a relatively small amount of TD. Hence, a critical question that deserves further investigation is the extent to which these tools agree upon the assessments of TD for a given set of classes.

[42] Although 3 tools have been used, the theoretical presentation of our approach is generalized for p tools

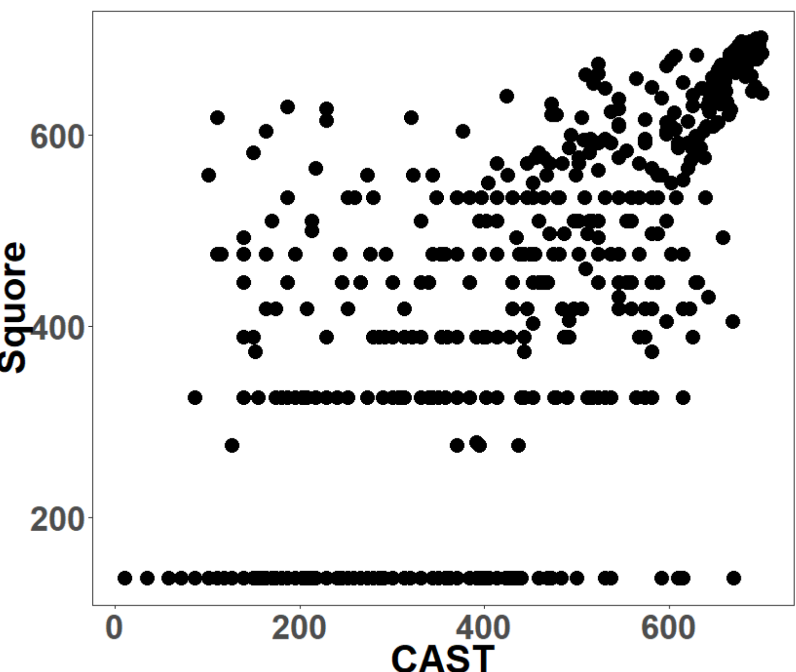


Figure 2. Scatter plot for rankings of TD measurements of opennlp project as evaluated by TD tools (CAST, Squore)

To this regard, a statistical measure, namely the Kendall's W coefficient of concordance [8] is employed, which belongs to the broader branch of methodologies known as inter-rater agreement analysis. In general, there is a plethora of measures for evaluating the agreement among assessors and the choice should be based on (i) the total number of assessors that assign to each subject a unique measurement (or rating), (ii) the scale of measurement (nominal with two or more categories, ordinal, continuous scale) that is assigned to each subject and (iii) the objectives of the analysis [26]. More specifically, the Scott's π [27] and Cohen's κ [28] are well-known measures for inter-rater agreement on a nominal dichotomous (No/Yes, Negative/Positive) scale that can be used in cases, where there are exactly two assessors. For the case of multiple assessors (more than 2) on nominal (either dichotomous or with multiple categories) or ordinal scales, the Fleiss's κ [29], which is a generalization of Scott's π coefficient and the weighted Cohen's κ [30] are possible choices that take into account not only the agreement but also the disagreement among them. All the aforementioned coefficients share the same rationale that is to evaluate and statistically test whether the average agreement between two (or more assessors) is significantly different than chance. An additional problem to the ordinal ratings, besides the fact that agreement and disagreement are no longer distinct notions [26], is the fact that there is another kind of agreement that may be of interest. This can be defined "as the agreement among raters with respect to the ranking of subjects" [26], which, in our case, is related to the process of evaluating whether all assessors, agree on which classes are the highly-ranked, the second highly-ranked and so on. In this case, the selection of the most appropriate agreement coefficients should belong to the branch of measures of concordance [26], since in general the variation of kappa statistics evaluate the absolute agreement between ratings, while concordance coefficients measure the association between ratings. Finally, a well-known limitation of kappa statistics is their dependence on the number of categories of the response measurement, since they tend to be generally higher, when there are fewer categories [31].

Summarizing, the choice of Kendall's W concordance coefficient instead of other kappa measures of agreement was based on the facts that (i) it serves in a straightforward manner the investigation of RQ1, which is related to the evaluation of the degree of agreement among TD measurement tools and (ii) it handles in an appropriate way the characteristics of the experimental design, which involves three TD assessment tools (CAST, Squore, SonarQube) and the derived rankings ranging from 1 up to n (the total number of the examined classes). Due to the existence of a high number of tied ranks in each tool (Figure 2), a modification of the original statistic that provides a correction for ties is employed. The Kendall's W statistic [32] is defined as

$$W = \frac{12\sum_{i=1}^{n} r_i^2 3p^2 n(n+1)^2}{p^2 n(n^2-1) - pT} \tag{1}$$

where, n is the total number of the examined classes, $\sum_{i=1}^{n} r_i^2$ is the sum of the squared sums of ranks for each of the n classes and p is the total number of the examined tools (three in our case). The term T is a correction factor for tied ranks that is evaluated via the following formula

$$T = \sum_{k=1}^{g} \left(t_k^3 - t_k\right) \tag{2}$$

in which, t_k is the number of tied ranks in each of g groups of ties, whereas the sum is evaluated over all groups of ties found in all p tools of Table 4. Kendall's W can take a range of values from 0 (indicating no agreement) to 1 (indicating a perfect agreement among assessors). In addition, Schmidt [33] provides specific guidance through rules of thumb on how researchers should interpret experimental results based on the evaluation of the Kendall's W statistic. More specifically, a coefficient of 0.7 or higher can be interpreted as a strong agreement among the set of assessors. For example, the evaluation of the Kendall's W concordance coefficient for the set of classes of our demonstrative example indicates a statistically significant strong agreement between the CAST and Squore tools regarding their TD assessments, $W = 0.874, p < 0.001$.

3.4.2. Benchmarking through Archetypal Analysis ($RQ_2 - RQ_4$)

From what was already mentioned, there are several available tools for assessing TD, whereas each tool is based on a different ruleset that may result to divergent TD assessments for a given project. Although, this fact could lead to the identification of alternative mitigation actions, the empirical evidence reveals that software practitioners and development teams usually base the measurement process of TD on a single tool. Having in mind that there is no ground truth for assessing TD, there is an imperative need for the empirical examination of the diversity produced by the utilization of a set of alternative TD tools. Indeed, the findings from the indicative example discussed in the previous section revealed that despite the fact that there is a strong agreement between the assessments provided by the two examined tools, the tools also disagree upon the measurement of TD of some classes.

Towards this direction, an agreement-based benchmark approach is proposed, contributing to the empirical characterization of the assessments provided by a set of p alternative tools with respect to the derived TD evaluations for a given set of n examined classes. The benchmark framework is based on a statistical approach, namely *Archetypal Analysis* (AA) [9]. Describing the general principles of the methodology, AA is a data-driven multivariate method that explores a multidimensional space of points (or observations) with the aim of identifying certain observations, namely the archetypes, located on the boundaries of a swarm of given points (or *convex hull*). An interesting property of the methodology is the fact that the swarm of points can be represented as convex combinations of the archetypes. The latter provides a straightforward

mechanism supporting the identification of a subset of points that are closer to a specific archetype, which in turn, can be used for benchmarking purposes.

In our context, the input for AA is the $n \times p$ matrix (Table 4) representing the rankings of TD assessments derived from the analysis conducted through the utilization of a set of p tools for a given project with n classes. The algorithm of AA seeks for a matrix Z of $k \times p$, where k and p are the number of archetypes and dimensions, respectively through the computation of two coefficient matrices a and b minimizing the residual sum of squares (RSS) defined as

$$\text{RSS} = \left\| X - \text{a}Z^T \right\|_2 \text{with } Z = X^T b \tag{3}$$

where $\| \quad \|_2$ denotes the Euclidean matrix norm, subject to the following constraints:

$$\textstyle\sum_{j=1}^{k} a_{\text{ij}} = 1 \text{ with } a_{\text{ij}} \geq 0 \text{ and } i = 1,\ldots,n \tag{4}$$

$$\textstyle\sum_{i=1}^{n} b_{\text{ji}} = 1 \text{ with } b_{\text{ji}} \geq 0 \text{ and } j = 1,\ldots,k \tag{5}$$

These constraints frame the two general properties of AA which are: (*i*) the approximated data (swarm of points) are convex combinations of the archetypes, i.e. $X = \text{a}Z^T$, and (*ii*) the archetypes are convex combinations of the data points, i.e. $Z = X^T b$. The term “convex combination” refers to the linear combination of points, when all coefficients are non-negative and their sum is equal to 1. Computationally, the algorithm reduces the RSS in Eq. (3) by iteratively calculating the archetypes along with the coefficient matrices a and b. Summarizing, the archetypal solution provides an approximation of the convex hull defined by the swarm of points in the multidimensional space through the evaluation of a few, not necessarily observed points, lying on the boundaries of the observed points.

Due to the intuitive rational and interesting properties of AA, the method has been widely used for benchmarking purposes in many scientific domains [34], e.g. such as marketing [35], astrophysics [36], sports analytics [37], biology [38], medicine [39], scientometrics and bibliometrics [40], multi-document summarization [41], neuroscience [42] etc. In Software Engineering, AA has been introduced in [43], [44], in which the objectives were the evaluation of the predictive capabilities of a set of Software Effort Estimation (SEE) models and the building of ensembles using a subset of inferior models, whereas in [45], the authors explored psychometric data in order to extract different software engineers profiles based on measurements from their personality and behavioral characteristics.

Following a similar approach to [46], in this dissertation, AA constitutes the core methodology of a three-step process that facilitates the examination of the diversity of TD assessments provided by a set of alternative tools with the aim of identifying a set of classes exhibiting similarity to a selected archetype that can be used, in turn, for benchmarking purposes. Such classes can, for example, be classes with increased levels of TD as measured by all three tools, or TD-clean classes, which present limited inefficiencies. The three basic steps of the proposed approach summarized into the following points constitute the basis of the methodology for providing answers to RQ2 - RQ4:

1. Identification of archetypes representing the reference assessments through the exploration of the diversity of TD assessments derived from the set of employed tools (RQ2).
2. Reification of archetypal solution into the context of TDM through the identification of their characteristics (RQ3).
3. Identification and retrieval of a set of classes that are close to archetypes depicting either high TD or low TD assessments as suggested by all employed tools (RQ4).

The implications of the three previous steps are clearly demonstrated through the application of the approach on the indicative example described in previous section. In Figure 3, the boundary of the grey area defines the convex hull of all TD assessments derived from the CAST and Squore tools through the examination of classes from opennlp project. Based on the principles of AA, the archetypes representing the reference assessments will lie on this boundary, whereas the shape of the convex hull provides straightforward answers regarding the diversity of the examined set of TD tools.

A critical decision that someone has to take is the selection of an appropriate number of the k archetypes that approximates the convex hull in an efficient way. Certainly, the number of archetypes plays a significant role to the efficient representation of the swarm of the observed points, since the diversity of the convex hull may be better captured, as the number of archetypes increases. In contrast, one has to take into consideration that an unnecessary large number of archetypes might not contribute further to the approximation of the convex hull, whereas it would also affect the benchmarking process, since the objective is the extraction of few reference assessments representing useful profiles of practical importance to both researchers and practitioners in TDM.

To this regard, the graphical inspection of the swarm of TD assessments (Figure 3) suggests that the efficient number of archetypes capturing the diversity of the two examined TD tools is $k = 4$ archetypes. In the trivial case of $k = 1$, the archetypal solution is the centroid of the two-dimensional space representing the TD assessments matrix (Table 4), whereas its coordinates are easily calculated by the univariate sample mean values of TD rankings from each tool (sample means of CAST and Squore TD columns in Table 5).

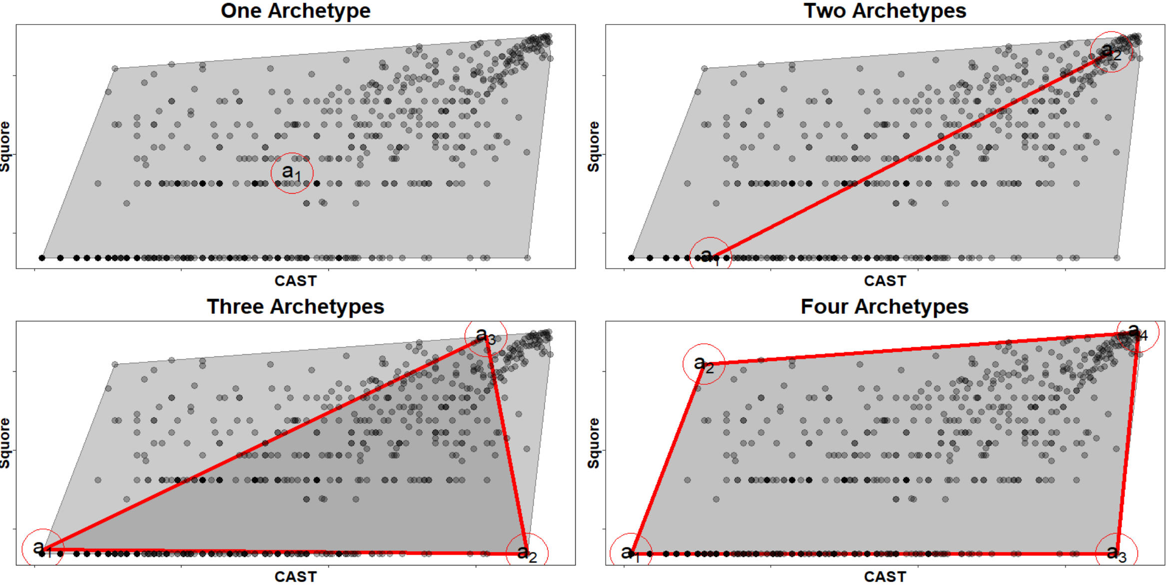


Figure 3. Archetypal solutions (CAST, Squore) for opennnlp project

Although the graphical inspection constitutes a straightforward manner for the identification of the appropriate number of archetypes in the special case of the two-dimensional space, i.e. the examination of assessment provided by two TD tools, this is not the case, when the number of the examined TD tools is higher than two (p>2). In order to provide certain guidelines about the decision upon the appropriate number of archetypes, Cutler and Breiman [9] suggest the utilization of the graphical inspection of the RSS reduction plot (or elbow plot). The RSS plot (Figure 4) constructed after consecutive executions of AA for

different values of k, $(k = 1,2,3,4,5)$ confirms our intuitive beliefs derived from the graphical inspection of the two-dimensional example. More specifically, considering that the line displaying the RSS reduction looks like an arm, then an elbow appears at $k = 4$, pointing out the optimal number of archetypes. The idea is that after this specific point ($k > 4$), the line flattens and hence, the extracted solution ($k = 5$) does not contribute to any further reduction of RSS. Summarizing, the practical implication of the first step (Step 1) of the proposed approach on the indicative example, is that four reference assessments (archetypes) can capture the diversity of TD rankings derived from the static code analysis (by two tools) for the set of the examined classes of opennlp project.

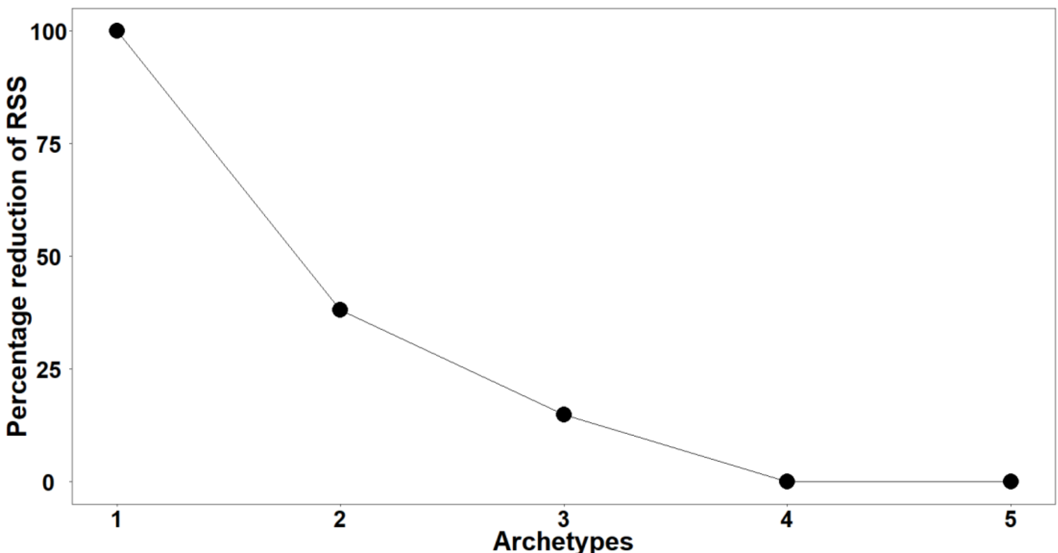


Figure 4. RSS plot (CAST, Squore) for opennnlp project)

In the second step (Step 2), the objective was to understand the characteristics of the derived archetypes with the aim of extracting information regarding their meaning from a practical point of view in TDM. The relative position of the four archetypal solutions (Figure 3) and the graphical examination of the profiles plot (Figure 5) provide a clear overview of what each archetype really represents. More specifically, the profiles plot shows the evaluated TD rankings (CAST and Squore coordinates, Figure 3) for each archetype of the final solution. In addition, it can also be observes that the examination of the characteristics provides also a semantic categorization of the derived archetypes into two distinct groups, which are (i) the *Ruler* and (ii) the *Rebel* archetypes[43] [47]. The former group (The Ruler) reifies a reference assessment profile, in which the two tools agree upon either on low (The Min-Ruler archetype a_1) or high (The Max-Ruler archetype a_4) TD rankings assessments. The latter group (The Rebels) reifies a reference assessment profile, in which the two tools do not agree on their TD rankings assessments signifying a completely divergent behavior of the two assessors. Overall, the four archetypes represent the following distinct reference assessment profiles with the following characteristics:

- **The Max-Ruler** (archetype a_4 in Figure 3d) represents the reference assessment corresponding to the profile of classes accumulating high amount of TD based on the results of both tools (CAST and Squore).

[43] The idea of archetypes was developed by psychologist C. Jung in his studies about drivers of human behavior. Pearson suggested the use of 12 archetypes among which the 'Ruler' denotes personalities whose goal is to *create a prosperous, successful family or community*, while for a 'Rebel' (also known as Outlaw) the motto is that *rules are made to be broken*. In our context, the 'Ruler' profile denotes a community of classes sharing the same assessment by all employed tools, while the 'Rebel' points to tools that in some sense break the rules and identify TD items in a different way than the rest.

- **The Min-Ruler** (archetype a_1 in Figure 3d) represents the reference assessment corresponding to the profile of classes accumulating low amount of TD based on the results of both tools (CAST and Squore).
- **The Rebel 1** (archetype a_2 in Figure 3d) represents the reference assessment corresponding to the profile of classes accumulating low amount of TD based on the results of the analysis from the CAST tool, but on the same time, high amount of TD based on the results of Squore tool.
- **The Rebel 2** (archetype a_3 in Figure 3d) represents the reference assessment corresponding to the profile of classes accumulating high amount of TD based on the results of the analysis from the CAST tool, but on the same time, low amount of TD based on the results of Squore tool.

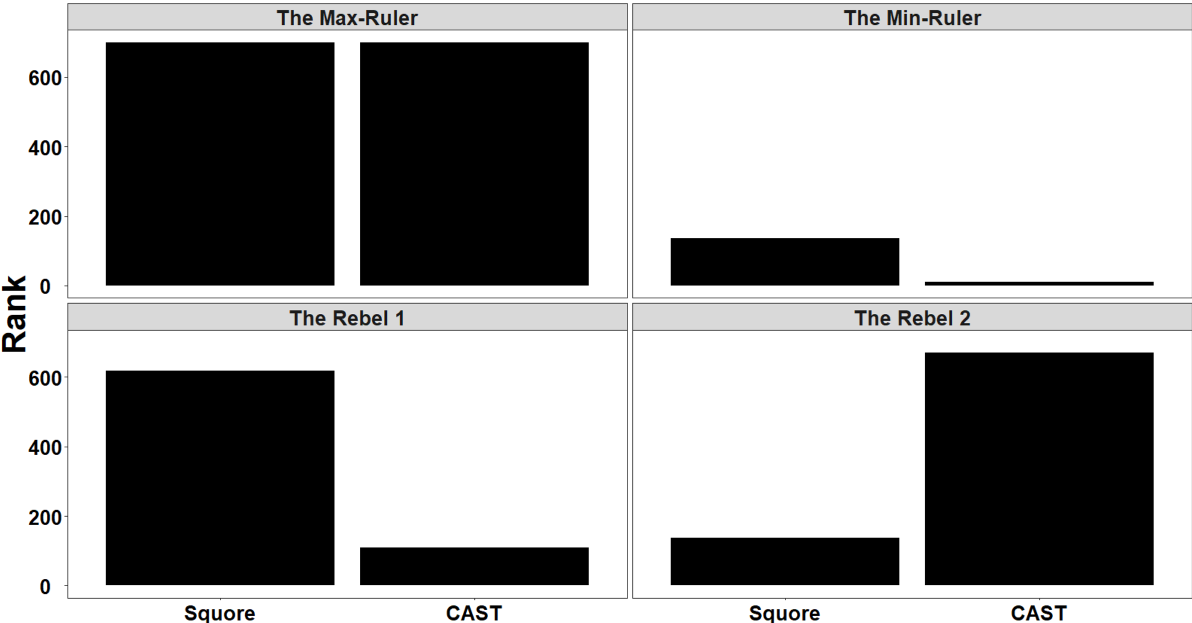


Figure 5. Reference assessment profiles (archetypes) (opennnlp project)

After the reification of the archetypes, the final step (Step 3) of the proposed approach involves the identification and retrieval of a set of classes that are close to a specific archetype gathering certain characteristics that can be used for TDM purposes. A critical challenge in the TD community raises from the fact that although there are several available tools for measuring and monitoring TD, the community has not concluded on a state-of-the-art solution that could be used as a ground truth for measuring TD. Developers and researchers acknowledge that TD estimates provided by any single tool are inherently subjective, reflecting a particular strategy for the identification of TD items. The existence of a basis of classes that are assessed as high TD modules by various tools would point to classes that can objectively be classified as validated high-TD modules and would boost relevant research. Currently, the lack of a commonly agreed way of quantifying TD impedes the development of approaches that could built on top of TD measurements, as in the case of machine learning approaches seeking to identify code or design problems employing alternative parameters as inputs. The ability to derive a benchmark of classes being close to the Max-Ruler archetype can be directly leveraged for training supervised learning-based algorithms. Similarly, the classes which have been validated as high-TD by all tools can be analyzed by development teams to seek non ideal coding practices and patterns so as to avoid them in future releases. On the other hand, benchmarks of classes formed by those that are close to Rebel archetypes essentially designate design or code inefficiencies which are captured by only one of the available tools, possibly

pointing to unique features identified by a particular ruleset. As a result, the union of classes belong to these sets would ensure the widest possible coverage of TD liabilities.

The evaluation of the adjacency of a certain TD assessment (representing a given class) to each archetype can be practically accomplished through the matrix of the α-coefficients (Eq. 4). More importantly, due to the first property of AA, i.e. the approximated points are convex combinations of the archetypes that are summed to unity, the computed α-coefficients for each TD assessment provide an easily interpretable mechanism for quantifying its resemblance to all archetypes. Table 5 displays the classes and their TD assessments that are close to the Max-Ruler archetype according to the threshold value of $\alpha = 0.80$ for characterizing the neighboring classes. By setting the threshold value of $\alpha = 0.80$, a set of 84 out of 701 total classes (almost 12% of the examined classes) can be considered as adjacent to the Max-Ruler archetype. This practically means that a practitioner has access to a set of classes that have been validated as high-TD classes by all tools. Due to space limitations, only the first and last five classes from the 84 are presented, which are close to the Max-Ruler archetype. Interpreting the vector of α-coefficients for a randomly selected class, e.g. C_{42} with ($\alpha_{\text{Min-Ruler}}$, $\alpha_{\text{Rebel 1}}$, $\alpha_{\text{Rebel 2}}$, $\alpha_{\text{Max-Ruler}}$) = (0.091, 0.000, 0.001, 0.908) (last four columns of Table 5), it can inferred that C_{42} is 9.1%, 0.0%, 1.0% and 90.8% similar to the Min-Ruler, Rebel 1, Rebel 2 and Max-Ruler archetypes, respectively, and for this reason it is considered as a neighboring class to the Max-Ruler archetype.

Finally, Figure 6 visualizes the neighbourhood of the Max-Ruler archetype (corresponding to the TD measurements of the abovementioned 84 classes) with a black-scaled colour indicating the degree of resemblance for each TD assessment to this specific reference assessment profile. Moreover, points denoted by empty red circles represent classes that are not similar to the Max-Ruler archetype ($\alpha < 0.80$) in terms of their TD assessments.

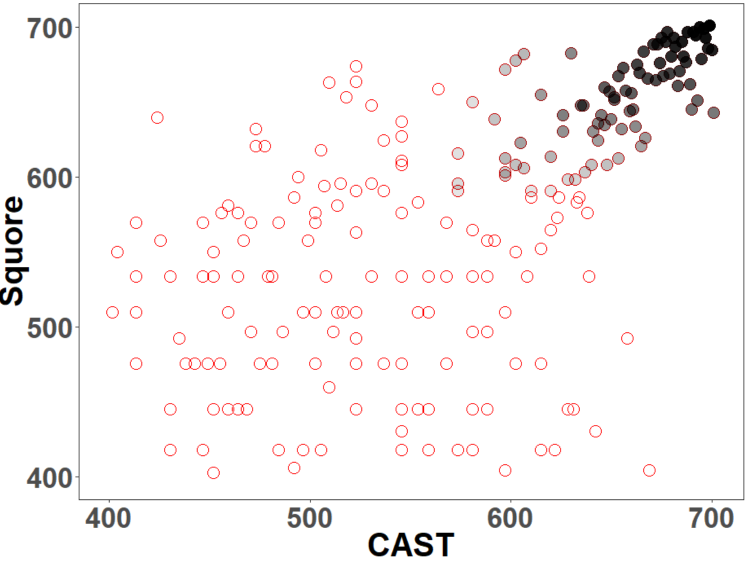


Figure 6. Scatter plot for neighboring classes to the Max-Ruler archetype (CAST, Squore) (opennnlp project)

Table 5. Indicative set of classes that are close to the Max-Ruler archetype (CAST, Squore) (opennnlp project)

Class		**Ranking**		**a-coefficient**			
ID	Name	Squore	CAST	The Min-Ruler	The Rebel 1	The Rebel 2	The Max-Ruler
C_1	/main/java/opennlp/tools/stemmer/snowball/turkishStemmer.java	701	699	0.000	0.000	0.000	1.000
C_2	/main/java/opennlp/tools/stemmer/snowball/englishStemmer.java	699	696	0.001	0.004	0.000	0.995
C_3	/main/java/opennlp/tools/stemmer/snowball/frenchStemmer.java	700	694	0.000	0.008	0.000	0.992
C_4	/main/java/opennlp/tools/stemmer/snowball/portugueseStemmer.java	695	692	0.008	0.002	0.000	0.989
C_5	/main/java/opennlp/tools/stemmer/snowball/hungarianStemmer.java	693	697	0.003	0.000	0.010	0.988
…	…	…	…	…	…	…	…
C_{42}	/main/java/opennlp/tools/formats/Conll03NameSampleStream.java	648	636	0.091	0.000	0.001	0.908
…	…	…	…	…	…	…	…
C_{80}	/main/java/opennlp/tools/formats/ontonotes/OntoNotesNameSampleStream.java	650	581	0.072	0.117	0.000	0.812
C_{81}	/main/java/opennlp/tools/ml/BeamSearch.java	616	573.5	0.142	0.047	0.000	0.811
C_{82}	/main/java/opennlp/tools/util/ObjectStreamUtils.java	591	573.5	0.182	0.000	0.012	0.807
C_{83}	/main/java/opennlp/tools/cmdline/namefind/TokenNameFinderTrainerTool.java	591	620	0.111	0.000	0.082	0.807
C_{84}	/main/java/opennlp/tools/lemmatizer/LemmatizerME.java	591	610	0.126	0.000	0.067	0.807

The final step of the benchmarking process is supported by a web application (TD Benchmarker) that has been developed which enables the extraction of benchmarks, consisting of classes being close to a selected archetype, for varying threshold values. Interested researchers can download the agreement-based benchmark of choice and retrieve the identified classes for further experimentation. Moreover, the application provides graphical illustrations of the RSS plots and the reference assessment profiles. TD Benchmarker is available online[44].

RQ1: For RQ1, where the level of agreement of the used tools is examined with respect to the measured TD of classes, the Kendall's W coefficient of concordance was employed which belongs to the broader branch of methodologies known as inter-rater agreement analysis.

For RQ2 – RQ4 an agreement-based benchmark process is proposed, which is based on a statistical approach, namely Archetypal Analysis (AA).

RQ2: In the first step of the benchmarking process, the aim is to calculate the required number of archetypes to effectively capture the diversity of the tools. In this regard, the appropriate number of archetypes was determined, via the graphical inspection of the RSS reduction plot (or elbow plot).

RQ3: In the second step of the benchmarking process the objective was to understand the characteristics of the derived archetypes in an attempt to interpret them from the Technical Debt Management (TDM) point of view. Through the graphical examination of the Archetypal Solutions figure two main categories of the archetypes were distinguished; the Ruler and the Rebel archetypes.

RQ4: The final step of the benchmarking process involves the identification and extraction of a set of classes that are close to a specific archetype with specific characteristics that can be interpreted in terms of TDM. The extraction of the aforementioned set of classes was accomplished through the matrix of α-coefficients (Eq. 4).

[44] https://se.uom.gr/index.php/projects/technical-debt-benchmarking

4. Results and Discussion

In this section the results for each research question are presented and discussed in the corresponding sub section.

4.1. RQ_1: To what extent do the assessors (tools) agree in the ranking of classes in terms of TD measurement?

Based on the proposed methodology (see Section 3.4), the objective is to investigate the degree of agreement among the applied TD tools (RQ1). Table 6 summarizes the results concerning the evaluation of the Kendall's W concordance coefficient for the set of the examined 50 projects. The results suggest that, in general, the three TD tools converge on the identification and measurement of TD at class/file level. Overall, the coefficient values range from 0.520 (for atom JavaScript project) to 0.853 (for javacv Java project). To this regard, it is meaningful to continue with the benchmarking process and extract the subset of classes which have been indicated as high-TD (or low-TD) classes by all tools. On the other hand, the graphical inspection of the aggregated results (Figure 7 (dot plots)) and the distributions of the coefficients for Java and JavaScript projects (Figure 8(a), (boxplots)) shows that the type of language seems to present an effect on the estimated agreement of TD tools. Indeed, an *independent-samples t-test* indicated a statistically significant difference between the mean values of Kendall's W concordance coefficient for Java ($M = 0.777$, $SD = 0.045$) and JavaScript ($M = 0.647$, $SD = 0.075$) projects, $t = 7.403$, $p < 0.001$ (Figure 8(b), (error bars)). *Levene's* test indicated unequal variances, $F = 7.628$, $p = 0.008$, so the t-test under the unequal variances assumption was used, whereas the *Kolmogorov-Smirnov* test for normality assumption showed that the estimated coefficients satisfied the normality assumption, K-S $Z = 0.893$, $p = 0.403$.

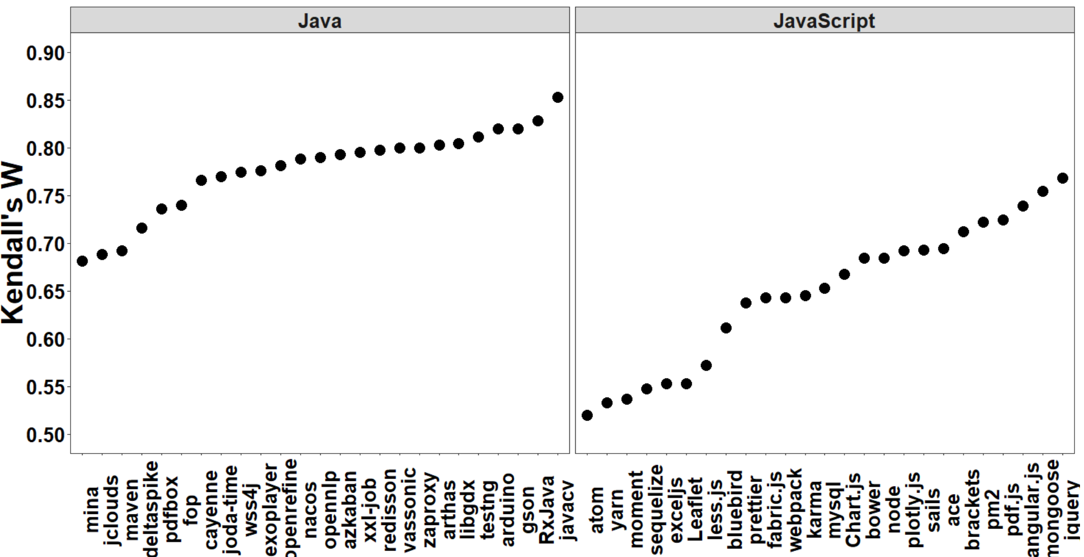
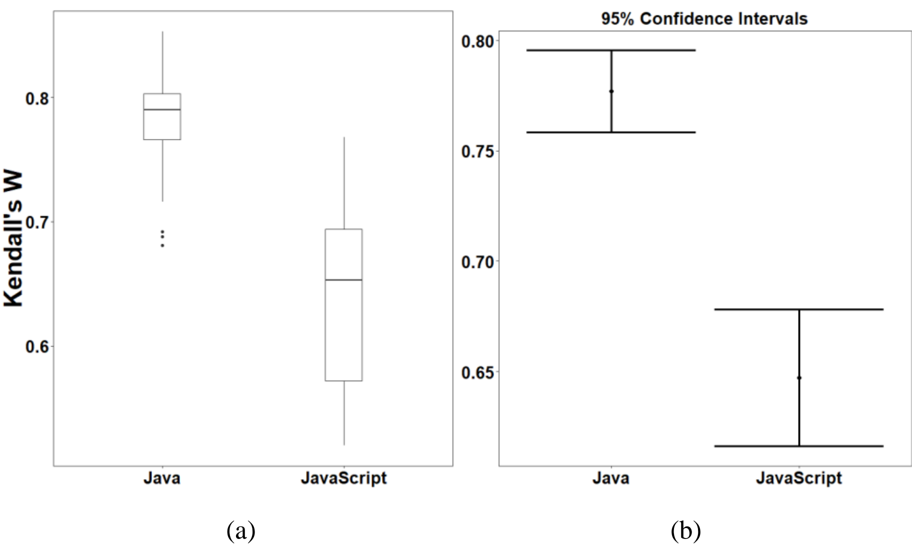


Figure 7. Dot plots with the aggregated results of Kendall's W concordance coefficient



(a) (b)

Figure 8. Box plots (a) and error bars (b) of the distributions of Kendall's W concordance coefficient

The general conclusion from the evaluation of the Kendall's W concordance coefficients and the rule of thumb proposed by Schmidt (1997) (see Section 3.4.1) is that in the case of Java projects, there is noted a statistically significant ($p < 0.001$) and strong agreement among the three tools regarding the TD assessments for the set of the conducted experiments with a mean value of 0.777 accompanied by a 95% CI ranging into the interval $[0.758, 0.795]$. In contrast, despite the fact that a statistically significant agreement among TD assessments is also indicated for the set of JavaScript projects, the strength of the agreement is characterized as moderate, since it presents a mean value of 0.647 with a 95% CI of $[0.616, 0.678]$. A possible interpretation for this finding is that tools for analyzing the quality of Java code (e.g. through static analysis) are more mature, compared to those for analyzing JavaScript, which are substantially younger. Therefore, it seems that along with their evolution Java analyzers have also converged on how the analysis is performed and what is deemed as an important problem for a codebase. On the other hand, it seems that JavaScript analyzers are in a more experimental stage, and therefore lower consensus is reached.

Table 6. Kendall's W Concordance Coefficient among all three TD tools for each analyzed system

Project	**W (*p*-value)**	**W (*p*-value)**	**W (*p*-value)**	**W (*p*-value)**	**W (*p*-value)**	**W (*p*-value)**	**W (*p*-value)**	**W (*p*-value)**	**W (*p*-value)**
Java									
arduino	0.820 (p <0.001)	exoplayer	0.776 (p <0.001)	joda-time	0.770 (p <0.001)	opennlp	0.790 (p <0.001)	testng	0.811 (p <0.001)
arthas	0.803 (p <0.001)	fop	0.740 (p <0.001)	libgdx	0.804 (p <0.001)	openrefine	0.781 (p <0.001)	vassonic	0.800 (p <0.001)
azkaban	0.793 (p <0.001)	gson	0.820 (p <0.001)	maven	0.692 (p <0.001)	pdfbox	0.736 (p <0.001)	wss4j	0.774 (p <0.001)
cayenne	0.766 (p <0.001)	javacv	0.853 (p <0.001)	mina	0.681 (p <0.001)	redisson	0.797 (p <0.001)	xxl-job	0.795 (p <0.001)
deltaspike	0.716 (p <0.001)	jclouds	0.688 (p <0.001)	nacos	0.788 (p <0.001)	RxJava	0.828 (p <0.001)	zaproxy	0.800 (p <0.001)
JavaScript									
ace	0.694 (p <0.001)	brackets	0.712 (p <0.001)	karma	0.645 (p <0.001)	mysql	0.653 (p <0.001)	prettier	0.637 (p <0.001)
angular.js	0.739 (p <0.001)	Chart.js	0.667 (p <0.001)	Leaflet	0.553 (p <0.001)	node	0.684 (p <0.001)	sails	0.693 (p <0.001)
atom	0.520 (p <0.001)	exceljs	0.553 (p <0.001)	less.js	0.572 (p <0.001)	pdf.js	0.724 (p <0.001)	sequelize	0.547 (p =0.002)
bluebird	0.611 (p <0.001)	fabric.js	0.643 (p <0.001)	moment	0.537 (p <0.001)	plotly.js	0.692 (p <0.001)	webpack	0.643 (p <0.001)
bower	0.684 (p <0.001)	jquery	0.768 (p <0.001)	mongoose	0.754 (p <0.001)	pm2	0.722 (p <0.001)	yarn	0.533 (p <0.001)

4.2. RQ2: How many archetypes (reference assessments) are required to capture the diversity of the tools?

After the verification of a statistically significant agreement among the three TD tools for the set of Java and JavaScript projects, the next challenge involves the benchmarking process with the aim to extract a set of classes identified as the most high-TD ones from all applied tools. Due to the extensive numerical and graphical results, the findings derived from the analysis (Step 1 - Step 3, see Section 3.4.2) are indicatively presented on opennlp project. Through this manner, it can be also highlighted to both researchers and practitioners how the proposed methodology can be easily generalized to any experimental setup without constraints regarding the number of applied TD tools. Finally, it should be reminded that the set of the experimental results along with the raw dataset of TD estimates for the 25 Java and 25 JavaScript projects can be easily accessed via the paper's web page[45].

Generalizing the methodology presented above (Section 3.4.2), the relative positions of the TD assessments via the three tools can be represented by a scatter plot in a three-dimensional space (Step 1). Figure 9 displays the TD assessments, in which each point represents again, a specific class with coordinates the TD rankings evaluated by the SonarQube (x-axis), CAST (y-axis) and Squore (z-axis) tools. Despite the fact that drawing conclusions from the inspection of a three-dimensional plot is not a straightforward task, the shape of the swarm of points reveals an intrinsic pattern. More precisely, there is a subset of classes that are concentrated on the upper left corner of the plot, corresponding to classes that accumulate a high amount of TD as it is assessed by the whole set of the applied tools. On the other hand, it is also obvious that there are also other regions on the graph indicating divergent behaviour of the applied tools in terms of their TD assessments. The practical implication of this phenomenon is that the three TD tools signify different mitigation actions, which is the consequence of the utilization of different rulesets in the evaluation process of TD.

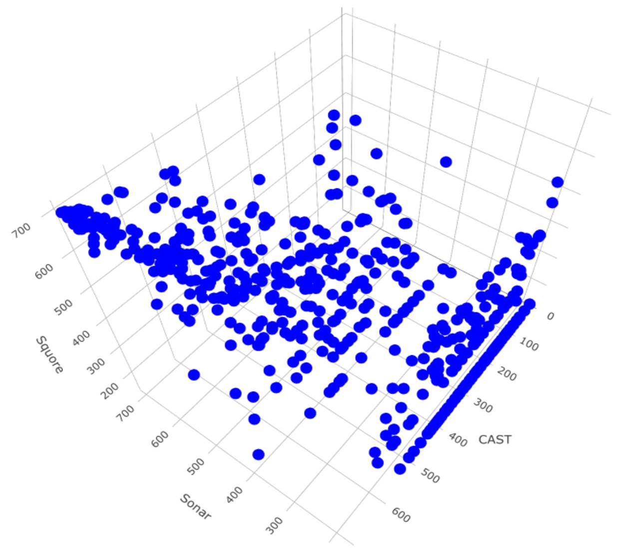


Figure 9. Scatter plot (3D) for the rankings of the TD assessments (all three tools) (opennlp project)

[45] https://se.uom.gr/index.php/projects/technical-debt-benchmarking

Indeed, the examination of the RSS (Figure 10) after the consecutive executions of the AA algorithm for different values of archetypes shows that the convex hull of the swarm of points can be adequately approximated by $k = 8$ archetypes. Generally, the examination of the RSS plots for the remaining datasets led us to conclude that this specific number of archetypes $k = 8$ is a rational generalization for the whole set of our experiments.

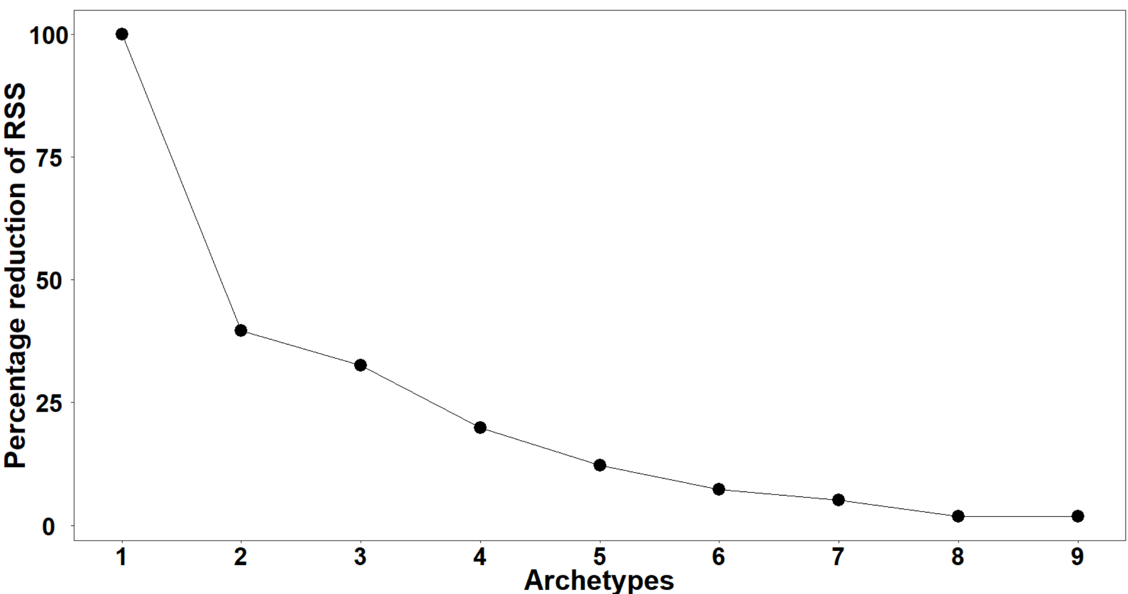


Figure 10. RSS plot (SonarQube, CAST, Squore) (opennnlp project)

4.3. RQ$_3$: Which are the characteristics of the extracted archetypes?

Having defined the appropriate number of archetypes ($k = 8$), the next step (Step 2) of the proposed approach concerns the reification of the extracted reference assessment profiles through the examination of their characteristics. Figure 11 summarizes the profile plots for each archetype of the derived solution. The examination of the characteristics of the eight profiles reveals, again, that there are two distinct groups (*Ruler* and *Rebel*) that have also been identified in the case of the TD assessments on the two-dimensional space (CAST and Squore) (see Section 3.4.2). Besides this fact, the analysis brings to the surface a new type of profile with specific characteristics regarding the assessments of the three tools. More specifically, the *Partner*[46] archetype represents a reference assessment profile, in which two of the applied tools indicate a high amount of TD, whereas on the same time, the third tool is not able to identify it indicating a low amount of TD.

[46] The Partner archetype refers to personalities whose goal is being in a relationship with people and surroundings. In analogy, the Partner profile in our case denotes cases where two of the three tools exhibit high agreement.

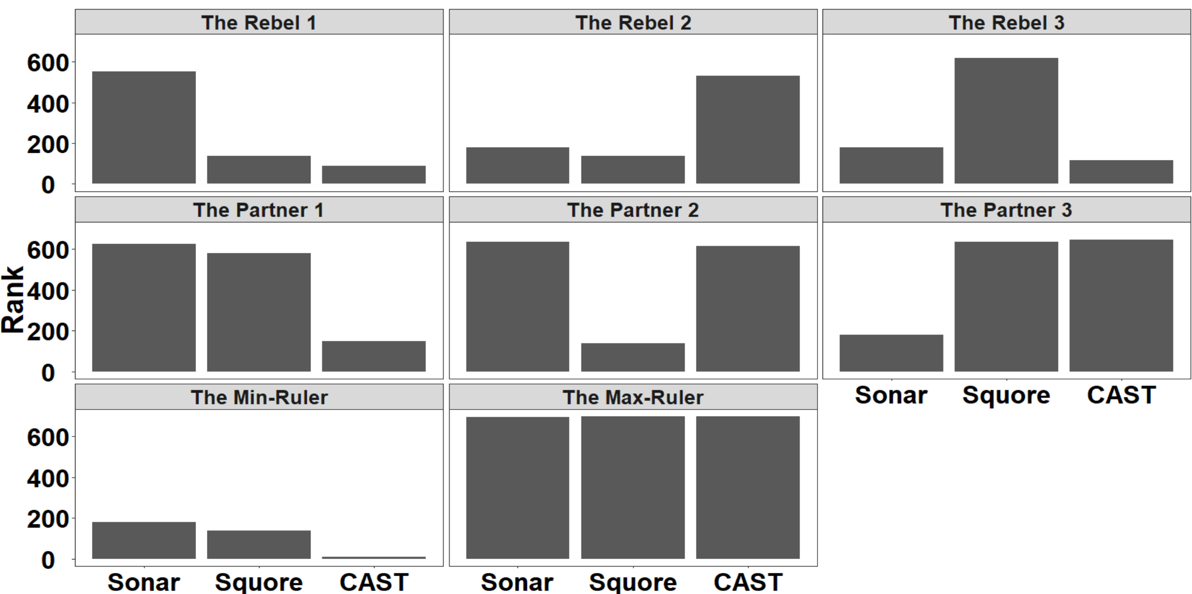


Figure 11. Reference assessment profiles (archetypes) from the assessments by all three tools (opennnlp project)

The characteristics of the $k = 8$ reference assessments (Figure 11) are fully described below:

- **The Max-Ruler** is the type of the reference assessment indicating a high amount of TD based on the results of all applied tools (SonarQube, CAST, Squore).
- **The Min-Ruler** is the type of the reference assessment indicating a low amount of TD based on the results of all applied tools (SonarQube, CAST, Squore).
- **The Partner 1** is the type of the reference assessment indicating a high amount of TD based on the results from SonarQube and Squore tools and simultaneously, a low amount of TD based on the results of CAST tool.
- **The Partner 2** is the type of the reference assessment indicating a high amount of TD based on the results from SonarQube and CAST tools and simultaneously, a low amount of TD based on the results of Squore tool.
- **The Partner 3** is the type of the reference assessment indicating a high amount of TD based on the results from Squore and CAST and tools and simultaneously, a low amount of TD based on the results of Sonar tool.
- **The Rebel 1** is the type of the reference assessment indicating a high amount of TD based on the results from SonarQube tool and simultaneously, a low amount of TD based on the results of Squore and CAST tools.
- **The Rebel 2** is the type of the reference assessment indicating a high amount of TD based on the results from CAST tool and simultaneously, a low amount of TD based on the results of SonarQube and Squore tools.
- **The Rebel 3** is the type of the reference assessment indicating a high amount of TD based on the results from Squore tool and simultaneously, a low amount of TD based on the results of SonarQube and CAST tools.

An interesting conclusion of the analysis on the remaining forty-nine datasets is that the abovementioned types of archetypes are applicable for the entire spectrum of projects and classes. It is reasonable to assume that the identified types of archetypes would be valid for any number of employed tools. For example, there will always be some classes identified as having high TD (or low TD) by all assessors (conforming to the

Max-Ruler or the Min-Ruler archetype). Nevertheless, the number of commonly identified high-TD (or low-TD) classes is expected to decrease with the number of tools. Similarly, it is also highly probable that one of the employed tools will tag some classes as high-TD while all other tools will not, according to the Rebel archetype, or that some subsets of tools might agree to a larger extent (Partners). This inherent trade-off should be considered by development teams when opting for particular quality assurance tools. The ‘intersection’ of commonly agreed artefacts with TD principal is expected to become lower as the number of tools increases and the benefit of obtaining wider coverage should be weighed against the diversity of the findings and the difficulties in incorporating multiple tools in the workflow. Practitioners and researchers should be assisted in focusing on the modules that are most likely to suffer from TD and to this end the next RQ aims at selecting the right set of classes for further analysis.

4.4. RQ_4: Which classes should be selected to form an agreement-based benchmark of top-TD modules?

In the last step of the methodology (Step 3), the focus is now on the identification of classes that are close to the archetype signifying top-TD classes as assessed by all tools. Practically, the target is classes settled in the neighborhood of the Max-Ruler archetype, which in turn can be specified through the definition of a threshold value for a coefficient. For example, in project opennlp, with $a = 0.80$ as a threshold value to capture a strong similarity (or adjacency) to the Max-Ruler archetype (in analogy to the 2-tool representative) example presented in Section 3.4.2), for three tools we would obtain 54 top-rated TD classes (7.70% of the total), while for two tools we obtained 84 top-rated TD classes (11.98%). The decrease in the number of commonly identified high-TD classes confirms the observation that the higher the number of assessors, the smaller the number of top-rated classes pointed out by all tools.

To examine the effect of the defined threshold value a on the percentage of top-rated classes extracted by the proposed approach, based on the source code analysis via the set of selected TD tools, sensitivity analysis was conducted. More precisely, the percentage of top-rated classes was evaluated for a set of threshold values of a-coefficients ranging from 0.60 to 0.90 increasing by a step of 0.05. In addition, there is an imperative need to investigate whether the type of language presents an effect on the percentages of top-rated classes for the above set of threshold values, since the inter-rater agreement analysis presented in Section 4.1 revealed a statistically significant effect of the type of language on the estimated concordance coefficients. Thus, an interesting issue that deserves further investigation is whether the type of language also affects the percentages of the top-rated classes.

Figure 12 summarizes the results from which, it can be generally inferred that the percentage of top-rated classes decreases as the threshold value increases for both language types. Practically, the selection of a higher threshold value imposes a stricter policy for the identification of high-TD classes by all employed tools. Another interesting finding is the fact that the percentages of top-rated classes/files seems to be generally higher for Java projects in comparison to JavaScript projects.

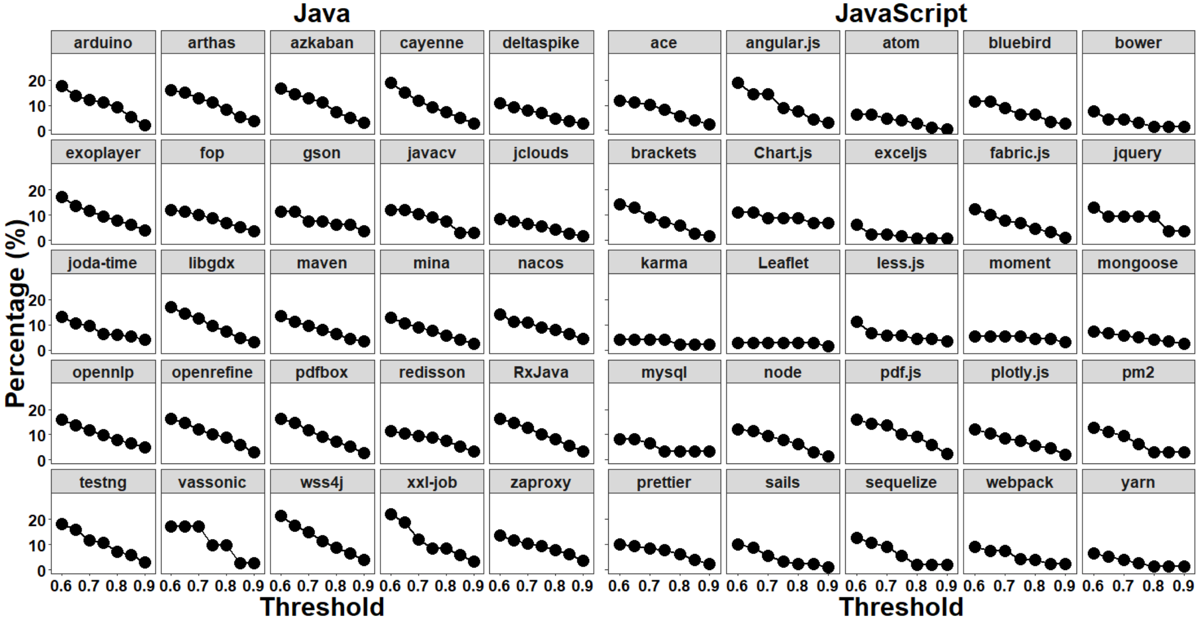


Figure 12. Percentage of top-rated classes assessed by all three tools for increasing levels of threshold values a (sensitivity analysis)

So, the next issue is to investigate, whether the observed phenomena can be generalized to the population of OSS projects with similar characteristics. For this reason, the *Linear Mixed Effects* (LME) models [48] are employed, which are able to model simultaneously two types of effects that are (*i*) the fixed effects, a term that is used to represent factors that may affect the mean value of interest, and (*ii*) the random effects that may have an impact only the variance of the response variable.

In this experimental setup, the *experimental unit*, for which we wish to draw conclusions regarding the response variable *Percentage* (i.e. the percentage of top-rated classes) is the project, which in fact, represents a unit drawn at random from an infinite unknown population of projects. For this reason, one should take into account and incorporate into the analysis, the random effect of the factor *Project*, in order to model the inherent variability caused by this random selection from the set of all possible OSS projects. Regarding the fixed effects that can been thought as the effect of specific factors of interest on the response *Percentage*, two factors need to be examined, which are (*i*) the threshold value (*Threshold*) of a denoting the closeness to the Max-Ruler archetype and (*ii*) the type of language (*Language*). Besides the abovementioned two *main effects* (*Threshold*, *Language*), there is also a need to examine the *interaction effect* of Threshold and Language (*Threshold*× *Language*), since the effect of the threshold value of a on the percentage of top-rated classes may not be the same at the two levels of language types (Java/JavaScript).

Regarding the fixed component structure, which describes the main and interaction terms that will be included in the inferential process, the optimal structure was defined through the protocol proposed by Zuur et al. [49]. Described briefly, a model (defined as the *beyond model*) examining all factors of interest and their possible interactions is fitted and tested against a second model after omitting the higher order interaction term through the *Likelihood Ratio* (LR) test. In case of an insignificant finding, the selection is based on the principle of parsimony, which practically means that simpler models with similar explanatory power are preferred over more complex models with more parameters but slightly better fit. To this end,

the *Akaike Information Criterion* (AIC) is used for the comparison process, while the model with the lowest AIC value should be preferred over the competitive ones.

The comparison of the beyond model (mentioned above) incorporating the main effects of *Threshold* and *Language* and their interaction term *Threshold*× *Language* against the model without the interaction term *Threshold*× *Language* did not reveal a statistically significant difference $\chi^2 = 6.055, p = 0.417$. The practical implication of this result is that the effect of the threshold value of a on the percentage of top-rated classes is the same for both language types (Java/JavaScript). The fitting of the final LME model containing only the main effects revealed statistically significant main effects for both *Threshold* ($F = 299.634, p < 0.001$) and type of *Language* ($F = 29.493, p < 0.001$) on the mean percentage values of top-rated classes. Is should also be noted that all models were fitted on the logarithmic transformations of the raw percentages, due to the violation of homoscedasticity assumption of model's residuals.

Moreover, the post-hoc analysis through Tukey's HSD test [48] for the factor *Threshold* indicates statistically significant differences ($p < 0.05$) between the pairs of consecutive levels of threshold values (as shown in Figure 13, the error bar does not cross the vertical dashed line of zero). Finally, Table 7 reports the expected mean percentage (accompanied by 95% CI) of top-rated classes for both language types in the population of OSS projects with similar characteristics in order to provide an indication of how many classes will be assessed as top-rated by all applied tools.

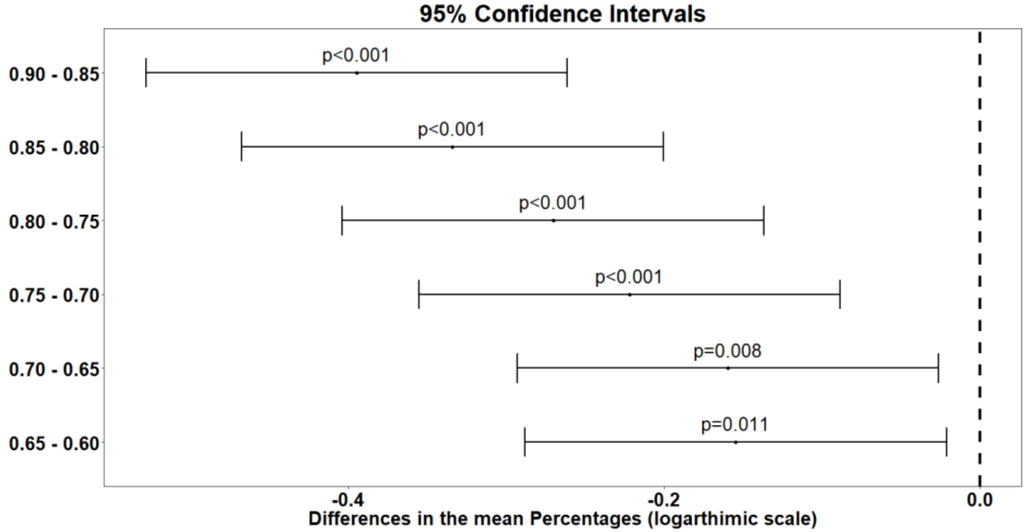


Figure 13. Post-hoc analysis for LME model (sensitivity analysis)

Table 7. Estimated mean percentage with 95% CI for each threshold value a (sensitivity analysis)

	Java		**JavaScript**	
Threshold	**Estimation**	**95 % CI**	**Estimation**	**95 % CI**
0.60	15.24	[13.17, 17.64]	9.11	[7.87, 10.55]
0.65	13.06	[11.28, 15.12]	7.81	[6.75, 9.04]
0.70	11.13	[9.62, 12.89]	6.65	[5.75, 7.70]
0.75	8.92	[7.71, 10.32]	5.33	[4.61, 6.17]
0.80	6.81	[5.88, 7.88]	4.07	[3.52, 4.71]
0.85	4.87	[4.21, 5.64]	2.91	[2.52, 3.37]
0.90	3.28	[2.84, 3.80]	1.96	[1.70, 2.27]

As it can be observed from Table 7, out of the total population of classes in each project and depending on the threshold value, only a small portion lying into the intervals [3.28%, 15.24%] and [1.96%, 9.11%] for Java and JavaScript projects, respectively, is characterized as having high-TD based on the findings of all three tools. Generally speaking, and without taking into consideration the type of language, this relatively low number of classes, in the neighborhood of the Max-Ruler archetype can be acknowledged as a basis concerning the high-TD classes. The resulting agreement-based benchmark can drive further research by denoting the few modules carrying "real-TD", rather than dealing with all candidates extracted by a single tool, which are not confirmed by other tools. Any future approach, leveraging also the power of machine learning, could be trained to accurately identify the top-rated classes capturing TD in a more realistic manner. It should be noted that a similar methodology could be applied for extracting a benchmark of low-TD classes. Such a set of classes might be valuable for studying the principles and practices resulting in cleaner code. Nevertheless, given the current priorities of development teams and researchers the focus was placed on benchmarks of high-TD classes. Besides the abovementioned findings, the analysis also indicates that irrespective to the applied threshold value a for characterizing similar classes to the Max-Ruler archetype, the percentages of the high-TD classes are expected to be higher for Java projects compared to the corresponding percentages derived from the analysis on JavaScript projects.

5. Implications to Practitioners and Researchers

In this section, the main outcomes are revisited, from the perspectives of practitioners and researchers. However, it should be borne in mind that any identified implications are subject to the limitations of the context in which the study has been performed. In particular, only the types of TD identified by the selected tools (namely design and code debt) and two programming languages (namely Java and JavaScript) have been considered. Moreover, the findings are based on a single measure of TD (i.e. principal) excluding other indices such as the severity or the type of the identified inefficiencies.

Overcome construct validity threats in research (researchers)

As mentioned in the introductory section, the research community within the TD field lacks an ultimate process to accurately capture TD principal and thus, any empirical study or technique based on TD estimates runs the risk of not accurately measuring the real-world phenomenon under study. Each tool follows its own approach for detecting and measuring TD, based on a distinct ruleset, yielding a different amount for the total TD, but also pointing to different parts of the code that need to be mitigated, compared to other tools. There are several studies trying to identify high-TD modules and studies investigating the association of accrued TD with other factors. However, such approaches are heavily dependent on the employed tool for suggesting the ground truth, that is, the modules that actually have TD liabilities and need to be fixed. Apparently, because each tool evaluates TD in a different way, the generalizability of these approaches is threatened to a large extent.

The two aspects of the proposed methodology, that is, the estimation of inter-rater agreement among TD tools and the use of archetypal analysis for identifying classes having a desired profile (e.g. high-TD levels by all tools) can be applied by researchers to form a more reliable basis for their experiments. More conveniently, researchers can also employ the already available benchmarks of high-TD classes (but also classes having a different profile if needed) from the online TD Benchmarker web application. Consequently, leveraging the power of multiple TD tools using the proposed approach can assist in the mitigation of construct validity threats that is currently present in the field of TD.

Highlight critical modules with validated highest TD (practitioners)

Despite the widespread adoption of the TD metaphor, it is far from clear which tool IT managers should integrate in the development and maintenance process. Employing more than one TD tool for the evaluation of their software might be a costly option, since most of the existing tools are available only with a commercial license. Moreover, each tool requires significant effort to deploy, properly configure and familiarize with. However, even if a development team employs more than one tool, the union of all findings, would result in an unrealistic amount of suggestions, rendering the process intractable. Based on the proposed methodology, practitioners can highlight the classes that have been identified as high-TD classes by all employed tools leading to a manageable number of target classes. Development teams can take advantage of such agreement-based benchmark sets and focus only on the modules of their system that are validated as high-TD modules. With respect to the benefit of the already derived benchmarks from the analyzed systems, developers can focus on the classes close to the Max-Ruler archetype and gain insight into the root causes of the accumulation of TD in these classes and potentially avoid non-ideal coding practices in the future. Moreover, the non-unanimous archetypes (Rebel and Partner archetypes) can be valuable, as well. The existence of these archetypes is the key factor that differentiates one tool from the others. If only unanimous archetypes existed, this would mean that all tools generate the same results pointing to the same classes/files with accrued TD principal. Through the exploration of classes/files in the vicinity of non-unanimous archetypes development teams can gain insight into how TD tools differ on the measurement and prioritization of TD principal. With such knowledge, developers can more confidently invest in the TD tool that best fits their perception of when a class/file is tagged as high-TD (or low-TD).

Collection of available TD tools (researchers and practitioners)

Last but not least, another contribution of the current work is the localization and collection of available TD assessment tools, as presented in Section 2. The list is by no means an exhaustive one, as numerous other tools offer functionality related to the identification of code smells, anti-patterns, rule violations, excessive metric values, etc. all of which are indicators of the existence of TD in software. Nevertheless, the presented tools can serve as starting point both for practitioners who are searching for a TD tool to integrate into their development process as well as researchers who are seeking an appropriate assessor of TD principal. In both cases, the proposed methodology can assist in the critical appraisal of the agreement or the diversity among tool findings.

6. Threats to Validity

This section presents and discusses potential threats to the validity of this case study, focusing on construct, reliability, and external validity [50], [51]. Internal validity is not considered, since causal relations have not been studied.

Construct Validity. Concerning construct validity, it can be argued that the basis of TD cannot be formed solely on the findings of TD assessment tools and as a result the study might inaccurately capture the actual phenomenon. The employed tools perform static source code analysis and thus the identified liabilities are primarily related to code TD, and in certain cases might also point to design or architectural problems. But according to the literature [23], [24] several other types of TD have been identified and might be present throughout all phases of the software development lifecycle, including Test, Documentation, Build, Infrastructure TD, etc. Consequently, the extracted TD measurements and the resulting benchmark represent only a portion of the system TD. However, code TD has been one of the mostly studied type of TD [24] and the target of most available tools, including the ones that have not been used in this dissertation. Furthermore, the steps of the proposed methodology are equally applicable to the findings regarding any

type of TD and thus benchmarks can be derived for other types of problems, provided that suitable measurement tools are available.

Another important threat to construct validity pertains to the exclusion from the study of other TD-related information, such as the specific type of the identified inefficiencies or their severity. Indeed, it might be the case that the level of agreement among tools varies depending on type/severity of issues and this warrants a further study. Although TD principal is an aggregate measure encompassing all kinds of identified problems, development teams would be more assured in case different tools agree on the more severe problems or the type of problems which they consider relevant to their software. Nevertheless, both aspects of the proposed approach for the quantification of the level of agreement among the tools and the extraction of representative archetypes can be applied to any subset of the identified TD issues.

Reliability. The described methodology outlines all steps followed to carry out the inter-rater agreement and archetypal analysis along with the provided web application that allows the extraction of benchmarks (sets of classes close to the Max-Ruler archetype) mitigates reliability threats. One potential threat to the ability of replicating this dissertation and reaching the same results is related to the optimal number of archetypes defined in Step 1 of the proposed approach (Section 3.4.2). The selection of the appropriate number of archetypes that is able to capture the diversity of the examined TD tools based on the inspection of the multidimensional space is, to some extent, a subjective process, especially in the case of a three-dimensional plot. In addition, the above visualization practice is not applicable in case the number of tools is higher than three. In these cases, the practitioner should base his/her choice on the examination of the profile plots and most importantly, on the inspection of the RSS plot to conclude on the appropriate number of archetypes. In this experimental setup, the investigation of these graphical manners led us to the definition of the optimal number of eight archetypes, which is a rational and common-sense finding, since the derived archetypes represent expected behaviors, in cases where three TD tools with partially different rulesets are used for benchmarking purposes.

External Validity. Regarding the external validity of the proposed approach, a potential threat to the generalization of the results is related to the identification and retrieval of the set of classes that are close to the Max-Ruler archetype (Step 3, Section 3.4.2), since the extracted set is certainly affected by the subjectivity and strictness of the practitioner. To this regard, a sensitivity analysis was conducted in order to examine how the choice of the threshold value for a-coefficient defining the neighbour classes affects the percentage of classes that belongs to the extracted benchmark set. Moreover, this work investigates the research questions in the context of 50 open source projects. Due to the limited number and types of the analyzed systems the conclusions regarding the observed level of agreement among the tools and the number of archetypes which are sufficient to capture the swarm of the observed points, probably cannot be generalized across other domains, programming languages or to proprietary software. A similar threat to external validity stems from the selection of TD assessment tools in the sense that this analysis was based on the identified violations, which in turn reflect the particular ruleset of each tool. Therefore, the findings on the agreement of TD assessment tools cannot be generalized beyond the employed tools.

7. Related Work

Since the first goal was to study the level of agreement among TD tools, this section presents previous studies that compare the techniques and results of tools that explicitly or implicitly measure TD. The second goal was to extract an agreement-based benchmark set of validated high-TD classes; therefore, other approaches to build such benchmarks or extract thresholds in the broader area of software maintenance are discussed, as well.

7.1. Comparison of Tools measuring Technical Debt

In a previous case study [52], the authors aimed at locating the architecture debts of a proprietary web portal system owned by a software outsourcing company using their own tool, *Titan*. The results of the Titan tool (TitanDebts) were compared to the results of the SonarQube tool (SonarDebts) that the company was already using. By examining the overlap between TitanDebts and SonarDebts, the authors found that ¼ of the total files (25 files) were found in the intersection of the most problematic files that Titan and SonarQube have identified. To this regard, the authors concluded that the Titan tool (which identifies architecture debts more effectively) and the SonarQube tool detect substantially different and complementary sets of files.

A case study in 2014 [53] compared four different techniques of TD evaluation (with the associated tool to run the analysis) including code smells (tool: codevizard), automatic static analysis issues (tool: FindBugs), grime build up and modularity violations (tool CLIO). The authors investigated whether the set of selected techniques/tools report the same set of modules as problematic and which was the overlap among them. The classes of 13 Hadoop releases were measured and 30 metrics were compared. The results of the study showed that the four techniques/tools had very little overlap, pointing to different problems in different modules.

In an experimental study [54], the authors investigated the correspondence between several technical debt estimation approaches and external software quality models. Specifically, they evaluated (a) SonarQubes's, (b) CAST's and (c) Marinescu's method [55] of technical debt estimation against the QMOOD quality model, which encompasses the quality attributes; reusability, flexibility, understandability, functionality, extendibility and effectiveness. They did not find evidence for strong relationship between the TD estimates and the quality attributes of the QMOOD model, except for one estimation method regarding only the flexibility and effectiveness quality attributes. The authors concluded that "*it is important that industry practitioners, ensure that the technical debt estimate they employ accurately depicts the effects of technical debt as viewed from their quality model*".

In a recent study [56], the authors, being motivated by the perception that design problems are more significant than coding errors for long-term software maintenance, aimed at investigating how three major TD tools (CAST, NDepend, SonarQube) capture design debt. Particularly, the authors distinguished the rules that capture design debt from a total of 466 examined rules from all three tools. Their results showed that all three tools mainly focus on non-design debt (only 19% of the rules captured design issues). Particularly, NDepend focuses the most on design rules (26% of its total rules are design-related), then follows CAST with 17% and SonarQube with 13%.

Fontana et al. (2016) examined the impact of the elimination of architectural problems in four Java projects on the quality indices of four tools (SonarQube, inFusion, Structural Analysis of Java (SA4J) and Structure101). The results showed that the architectural refactorings in the four examined systems did not have any impact on the SQALE index of SonarQube and as far as SA4J is concerned, its stability index was affected only in one system. Consequently, the authors concluded that the SQALE index of SonarQube and the stability index of SA4J are not capable of effectively capturing the notion of architectural debt.

In another study [58], the authors compared the techniques of five tools (CAST, inFusion, Sonargraph, SonarQube and Structure101) that provide some kind of Technical Debt Index (TDI). The comparison of the tools showed that all tools except for SonarQube exploit architectural information to form their TDIs. Moreover, two of the tools (inFusion and Structure101) do not calculate the cost for TD remediation (TD principal) whilst they only calculate the cost of keeping the software as it is (TD interest). On the other hand, CAST and SonarQube calculate only TD Principal and not TD interest. As far as the output

measurement, CAST and Sonargraph output cost in terms of US dollars, SonarQube in terms of time to remedy issues, while the rest produce either abstract values or values that are not expressed in money or time.

According to the abovementioned studies that compared different TD measurement tools, the results of each tool diverged from the results of the others. This phenomenon emphasizes our motivation to compare the TD estimates of several TD tools and extract the high-TD modules as identified by the tools altogether. It should be also noted that the aforementioned studies employed tools that measure TD either explicitly (generating a direct Technical Debt Index) or implicitly (generating a general quality index). Nevertheless, it should be reminded that, only tools that explicitly output a Technical Debt Index were employed to allow for more focused and direct comparison of the results on TD measurement.

7.2. Benchmarks in Software Maintenance

Several studies attempt to establish benchmark datasets so that software quality assessment approaches can be compared against them. Quite often the related research effort aims at building benchmarks to extract representative thresholds for source code metrics or quality indices, which can then serve as baseline for comparison with actual values of the systems under evaluation. A notable example of such benchmarks is the benchmark repository of Software Improvement Group (SIG) against which any selected system can be compared in terms of code quality and maintainability [59]. Below follows an overview of studies, in which the authors developed benchmarks and aimed at deriving thresholds for the evaluation of software quality (in descending chronological order).

In a recent study [60], the authors defended the idea that the extraction of metric thresholds should be tailored to each software domain. They collected a large set of 3107 Java systems across 15 domains from GitHub[47] and measured a set of 8 source code metrics with the CK Tool[48]. The aforementioned metrics reflected size, complexity and inheritance aspects of software. Then, the authors derived metric thresholds using the method supported by TDTool [61]. In particular, thresholds have been selected so as to represent various groups (i.e. high-90% and very high-95%) of the sorted metric values. The authors found evidence that "*metric thresholds vary across domains and most domain-specific thresholds differ from generic thresholds*".

Döhmen et al. (2016) built a benchmark for maintainability evolution with data from approximately 1750 industrial software systems. The data was collected from the Software Analysis Warehouse (SAW), a property of the Software Improvement Group (SIG). SAW contains the results of the software quality analyses that SIG conducts. The study focused on the production source code of the projects excluding testing and auto-generated code. The authors created a prototype of a benchmark for maintainability evolution. The benchmark was based on a group of systems, which were close to a selected open source system, *Crawljax*, in terms of maintainability and volume. The authors, first, selected the systems which had the 5% closest maintainability transitions to Crawljax and then, with the use of Empirical Cumulative Distribution Function (ECDF) found the systems that developed equal or worse than the compared system.

Comparison against existing systems has also been used as a method for assessing the software quality of a commercial system, property of an international company in the logistics domain [63]. The system was analyzed in terms of size, complexity, modularity, redundancy and technical debt with the utilization of SonarQube and NDepend. To evaluate the quality of the system, the author compared it with the quality of a set of 1892 open source projects from GitHub of similar age and programming language. The author

[47] https://github.com/
[48] https://github.com/mauricioaniche/ck

calculated the metrics of each project with SonarQube and then extracted the percentile thresholds of the metrics with RTTool [64]. The system's metric was considered "normal" if its value was near the middle percentiles and vice versa. The aforementioned benchmark was applied at file and at system level with aggregated values.

The notion of balance between real and ideal software design was used in a study in 2014 [65], in which the authors described a method for deriving relative thresholds for source code metrics. The method was based on evidence that source code metrics follow fat-tailed distributions, meaning that there is no typical value for them [66]. Therefore, the authors suggested that it is acceptable for some metrics not to follow absolute thresholds. To this regard, they proposed the concept of relative thresholds for evaluating source code metrics, where a percentage of source code entities should have values lower than an upper limit, whilst another percentage of entities is accepted to exceed upper limit due to specific requirements. The method was evaluated by applying it on the classes of 106 Java systems and extracting thresholds for seven metrics.

A benchmark-oriented calculation of TD was proposed by Mayr et al. (2014). Their benchmark-based model for calculation of Remediation Costs of software combined features from three existing TD calculation approaches; CAST model, SQALE model and the SIG model. Measures obtained with these models were normalized in terms of lines of code before used in the proposed model. For each metric, the authors calculated a quartile-based distribution dividing the normalized values of the metric in four areas. Metrics with values that laid below the lower or above the upper areas were considered non-conforming to the benchmark dataset. Ultimately, the authors tested their model by applying it on two open source projects, the quality of which had been previously evaluated and compared against the benchmark database. The experiment showed that the model was able to calculate remediation costs that reflected the relative (to the benchmark database) quality of the projects.

In another study [68], a method for extracting metric thresholds from benchmark data was designed. The method was applied on a benchmark of 100 C# and Java systems proprietary and open-source from a broad range of domains. The metrics were extracted for every entity of the system (method and file level) and were normalized with the weight of the entity. As weight of the entity, its size in terms of LOC was considered. Then the normalized metrics were placed in percentiles, from which the thresholds derived. Their contribution to the industry was to successfully use the thresholds derived with their methodology instead of the thresholds based on experts' opinion.

8. Conclusions and Future Work

The Technical Debt metaphor successfully captures, in monetary terms, the penalty that has to be paid because of shortcuts during software development. These shortcuts are known to introduce architectural, design and code inefficiencies in software systems and various TD tools aim at identifying them by testing the source code against specific rulesets. However, TD tools provide different estimates of TD principal pointing to different mitigation actions. These discrepancies make a lot of people in academia and practice skeptical about the validity of existing TD tools and hinder the further development of TD research as no ground truth for accurate TD instances can be established.

To address these limitations in the TD community an empirical study was performed whose goal was twofold: (a) to determine the level of agreement among three well-known TD tools and (b) build agreement-based benchmarks of high-TD classes/files from a dataset resulting from 50 open-source projects. Inter-rater agreement has been assessed, using Kendall's W coefficient of concordance. To capture the diversity of the examined tools with the aim of identifying representative class profiles archetypal analysis (AA) was

conducted. Once the derived reference assessments are characterized, it is straightforward to extract sets of classes exhibiting similarity to a selected profile (e.g. that of high TD levels in all employed tools) and in this way establish a basis.

The findings of the inter-rater agreement analysis suggest that there is a statistically significant and strong agreement among the three TD tools on the measurement of TD at class level. However, a substantial degree of disagreement has also been observed for the measured TD level for numerous classes. The application of the archetypal analysis revealed that three types of reference assessments can successfully capture the spectrum of TD measurements provided by three tools: One set of archetypes represents classes identified as high-TD modules by only one of the tools, the second profile encompasses classes for which two of the tools agree on the measured TD level, while the final type of archetype signifies a high amount (or low amount) of TD based on the results of all applied tools. Selecting the classes in the vicinity of the latter archetype yields an agreement-based benchmark of classes tagged as high-TD by all tools. Such benchmarks, beyond their value as fields of study for poor development practices that led to low quality classes, can potentially form the basis for training more sophisticated TD identification and measurement approaches.

The goal was to shed light into the level of agreement among TD tools and to establish a process for deriving an agreement-based benchmark set of high/low TD artifacts. Any interpretation of the results considering different perspectives, such as development context, role of developers (tester, designer, analyzer, etc.) was beyond the scope of this dissertation. Nevertheless, this forms a really interesting area of future work. Another interesting line of research would be to investigate to which degree TD tools are compliant with the guidelines of the OMG Specification on Automated Technical Debt Measure[49].

Finally, the nature of the examined rules by each tool might be a decisive factor for the TD principal estimates per class/file. Drilling down to the level of individual rule violations which are detected by each tool, can shed light into the cause of their agreement or discrepancy. One interesting line of further research would be to conduct such a study to investigate the similarity among the examined rules by mapping the rules adopted by each tool to the rules employed by the other tools.

References

[1] W. Cunningham, “The WyCash Portfolio Management System,” in *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum)*, New York, NY, USA, 1992, pp. 29–30, doi: 10.1145/157709.157715.

[2] T. DeMarco, *Controlling Software Projects: Management, Measurement, and Estimates*, 1 edition. Englewood Cliffs, N.J: Prentice Hall, 1986.

[3] J. M. Conejero *et al.*, “Early evaluation of technical debt impact on maintainability,” *J. Syst. Softw.*, vol. 142, pp. 92–114, Aug. 2018, doi: 10.1016/j.jss.2018.04.035.

[4] C. Izurieta, A. Vetrò, N. Zazworka, Y. Cai, C. Seaman, and F. Shull, “Organizing the technical debt landscape,” in *2012 Third International Workshop on Managing Technical Debt (MTD)*, Jun. 2012, pp. 23–26, doi: 10.1109/MTD.2012.6225995.

[5] A. Nugroho, J. Visser, and T. Kuipers, “An empirical model of technical debt and interest,” in *Proceedings of the 2nd Workshop on Managing Technical Debt*, Waikiki, Honolulu, HI, USA, May 2011, pp. 1–8, doi: 10.1145/1985362.1985364.

[49] https://www.omg.org/spec/ATDM/About-ATDM

[6] M. Nayebi *et al.*, "A Longitudinal Study of Identifying and Paying Down Architecture Debt," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, May 2019, pp. 171–180, doi: 10.1109/ICSE-SEIP.2019.00026.
[7] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter, "Tricorder: Building a Program Analysis Ecosystem," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, May 2015, vol. 1, pp. 598–608, doi: 10.1109/ICSE.2015.76.
[8] M. G. Kendall, *Rank correlation methods*. Oxford, England: Griffin, 1948.
[9] A. Cutler and L. Breiman, "Archetypal Analysis," *Technometrics*, vol. 36, no. 4, pp. 338–347, Nov. 1994, doi: 10.1080/00401706.1994.10485840.
[10] A. Martini and J. Bosch, "An Empirically Developed Method to Aid Decisions on Architectural Technical Debt Refactoring: AnaConDebt," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, May 2016, pp. 31–40.
[11] B. Curtis, J. Sappidi, and A. Szynkarski, "Estimating the Principal of an Application's Technical Debt," *IEEE Softw.*, vol. 29, no. 6, pp. 34–42, Nov. 2012, doi: 10.1109/MS.2012.156.
[12] A. Tornhill, "Assessing Technical Debt in Automated Tests with CodeScene," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Apr. 2018, pp. 122–125, doi: 10.1109/ICSTW.2018.00039.
[13] J. Holvitie and V. Leppänen, "DebtFlag: Technical Debt Management with a Development Environment Integrated Tool," in *Proceedings of the 4th International Workshop on Managing Technical Debt*, Piscataway, NJ, USA, 2013, pp. 20–27, Accessed: May 29, 2019. [Online]. Available: http://dl.acm.org/citation.cfm?id=2663297.2663301.
[14] S. Arvedahl, "Introducing Debtgrep, a Tool for Fighting Technical Debt in Base Station Software," in *Proceedings of the 2018 International Conference on Technical Debt*, New York, NY, USA, 2018, pp. 51–52, doi: 10.1145/3194164.3194183.
[15] L. Xiao, Y. Cai, and R. Kazman, "Titan: a toolset that connects software architecture with quality analysis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Hong Kong, China, Nov. 2014, pp. 763–766, doi: 10.1145/2635868.2661677.
[16] L. Xiao, Y. Cai, and R. Kazman, "Design rule spaces: a new form of architecture insight," in *Proceedings of the 36th International Conference on Software Engineering*, Hyderabad, India, May 2014, pp. 967–977, doi: 10.1145/2568225.2568241.
[17] K. Chopra and M. Sachdeva, "EVALUATION OF SOFTWARE METRICS FOR SOFTWARE PROJECTS," *Int. J. Comput. Technol.*, vol. 14, no. 6, pp. 5845–5853, Apr. 2015, doi: 10.24297/ijct.v14i6.1915.
[18] G. A. Campbell and P. P. Papapetrou, *SonarQube in Action*, 1st ed. Greenwich, CT, USA: Manning Publications Co., 2013.
[19] B. Baldassari, "SQuORE: a new approach to software project assessment," Aug. 2013.
[20] L. B. Foganholi, R. E. Garcia, D. M. Eler, R. C. M. Correia, and C. O. Junior, "Supporting Technical Debt Cataloging with TD-Tracker Tool," *Adv Soft Eng*, vol. 2015, pp. 4:4–4:4, Jan. 2015, doi: 10.1155/2015/898514.
[21] C. Fernández-Sánchez, H. Humanes, J. Garbajosa, and J. Díaz, "An Open Tool for Assisting in Technical Debt Management," in *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Aug. 2017, pp. 400–403, doi: 10.1109/SEAA.2017.60.
[22] T. S. Mendes, F. G. S. Gomes, D. P. Gonçalves, M. G. Mendonça, R. L. Novais, and R. O. Spínola, "VisminerTD: a tool for automatic identification and interactive monitoring of the evolution of technical debt items," *J. Braz. Comput. Soc.*, vol. 25, no. 1, p. 2, Jan. 2019, doi: 10.1186/s13173-018-0083-1.

[23] N. S. R. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman, “Identification and management of technical debt: A systematic mapping study,” *Inf. Softw. Technol.*, vol. 70, pp. 100–121, Feb. 2016, doi: 10.1016/j.infsof.2015.10.008.

[24] Z. Li, P. Avgeriou, and P. Liang, “A systematic mapping study on technical debt and its management,” *J. Syst. Softw.*, vol. 101, pp. 193–220, Mar. 2015, doi: 10.1016/j.jss.2014.12.027.

[25] R. van Solingen, V. Basili, G. Caldiera, and H. D. Rombach, “Goal Question Metric (GQM) Approach,” in *Encyclopedia of Software Engineering*, American Cancer Society, 2002.

[26] K. L. Gwet, *Handbook of Inter-Rater Reliability: The Definitive Guide to Measuring the Extent of Agreement Among Raters*, 4 edition. Gaithersburg, MD: Advanced Analytics, LLC, 2014.

[27] W. A. Scott, “Reliability of Content Analysis: The Case of Nominal Scale Coding,” *Public Opin. Q.*, vol. 19, no. 3, pp. 321–325, 1955.

[28] J. Cohen, “A coefficient of agreement for nominal scales,” *Educ. Psychol. Meas.*, vol. 20, pp. 37–46, 1960, doi: 10.1177/001316446002000104.

[29] J. L. Fleiss, “Measuring nominal scale agreement among many raters,” *Psychol. Bull.*, vol. 76, no. 5, pp. 378–382, 1971, doi: 10.1037/h0031619.

[30] J. Cohen, “Weighted kappa: Nominal scale agreement provision for scaled disagreement or partial credit,” *Psychol. Bull.*, vol. 70, no. 4, pp. 213–220, 1968, doi: 10.1037/h0026256.

[31] P. F. Watson and A. Petrie, “Method agreement analysis: A review of correct methodology,” *Theriogenology*, vol. 73, no. 9, pp. 1167–1179, Jun. 2010, doi: 10.1016/j.theriogenology.2010.01.003.

[32] N. J. Salkind, Ed., *Encyclopedia of Research Design*, 1 edition. Thousand Oaks, Calif: SAGE Publications, Inc, 2010.

[33] R. C. Schmidt, “Managing Delphi Surveys Using Nonparametric Statistical Techniques*,” *Decis. Sci.*, vol. 28, no. 3, pp. 763–774, Jul. 1997, doi: 10.1111/j.1540-5915.1997.tb01330.x.

[34] J. Moliner and I. Epifanio, “Robust multivariate and functional archetypal analysis with application to financial time series analysis,” *Phys. Stat. Mech. Its Appl.*, vol. 519, pp. 195–208, Apr. 2019, doi: 10.1016/j.physa.2018.12.036.

[35] S. Li, P. Wang, J. Louviere, and R. Carson, “ARCHETYPAL ANALYSIS: A NEW WAY TO SEGMENT MARKETS BASED ON EXTREME INDIVIDUALS,” p. 6, 2003.

[36] B. H. P. Chan, D. A. Mitchell, and L. E. Cram, “Archetypal analysis of galaxy spectra,” *Mon. Not. R. Astron. Soc.*, vol. 338, no. 3, pp. 790–795, Jan. 2003, doi: 10.1046/j.1365-8711.2003.06099.x.

[37] M. J. A. Eugster, “Performance Profiles based on Archetypal Athletes,” *Int. J. Perform. Anal. Sport*, vol. 12, no. 1, pp. 166–187, Apr. 2012, doi: 10.1080/24748668.2012.11868592.

[38] J. C. Thøgersen, M. Mørup, S. Damkiær, S. Molin, and L. Jelsbak, “Archetypal analysis of diverse Pseudomonas aeruginosatranscriptomes reveals adaptation in cystic fibrosis airways,” *BMC Bioinformatics*, vol. 14, no. 1, p. 279, Sep. 2013, doi: 10.1186/1471-2105-14-279.

[39] Elze Tobias, Pasquale Louis R., Shen Lucy Q., Chen Teresa C., Wiggs Janey L., and Bex Peter J., “Patterns of functional vision loss in glaucoma determined with archetypal analysis,” *J. R. Soc. Interface*, vol. 12, no. 103, p. 20141118, Feb. 2015, doi: 10.1098/rsif.2014.1118.

[40] C. Seiler and K. Wohlrabe, “Archetypal scientists,” *J. Informetr.*, vol. 7, no. 2, pp. 345–356, Apr. 2013, doi: 10.1016/j.joi.2012.11.013.

[41] E. Canhasi and I. Kononenko, “Weighted archetypal analysis of the multi-element graph for query-focused multi-document summarization,” *Expert Syst. Appl.*, vol. 41, no. 2, pp. 535–543, Feb. 2014, doi: 10.1016/j.eswa.2013.07.079.

[42] A. Tsanousa, N. Laskaris, and L. Angelis, “A novel single-trial methodology for studying brain response variability based on archetypal analysis,” *Expert Syst. Appl.*, vol. 42, no. 22, pp. 8454–8462, Dec. 2015, doi: 10.1016/j.eswa.2015.06.058.

[43] N. Mittas, V. Karpenisi, and L. Angelis, "Benchmarking Effort Estimation Models Using Archetypal Analysis," in *Proceedings of the 10th International Conference on Predictive Models in Software Engineering*, New York, NY, USA, 2014, pp. 62–71, doi: 10.1145/2639490.2639502.
[44] N. Mittas and L. Angelis, "Data-driven benchmarking in software development effort estimation: The few define the bulk," *J. Softw. Evol. Process*, vol. n/a, no. n/a, p. e2258, 2020, doi: 10.1002/smr.2258.
[45] M. V. Kosti, R. Feldt, and L. Angelis, "Archetypal personalities of software engineers and their work preferences: a new perspective for empirical studies," *Empir. Softw. Eng.*, vol. 21, no. 4, pp. 1509–1532, Aug. 2016, doi: 10.1007/s10664-015-9395-3.
[46] G. C. Porzio, G. Ragozini, and D. Vistocco, "On the use of archetypes as benchmarks," *Appl. Stoch. Models Bus. Ind.*, vol. 24, no. 5, pp. 419–437, 2008, doi: 10.1002/asmb.727.
[47] C. S. Pearson, *Awakening the Heroes Within: Twelve Archetypes to Help Us Find Ourselves and Transform Our World*, First Edition, First Pinting edition. San Francisco: HarperOne, 2015.
[48] J. Pinheiro and D. Bates, *Mixed-Effects Models in S and S-PLUS*. New York: Springer-Verlag, 2000.
[49] A. Zuur, E. N. Ieno, N. Walker, A. A. Saveliev, and G. M. Smith, *Mixed Effects Models and Extensions in Ecology with R*. New York: Springer-Verlag, 2009.
[50] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empir. Softw. Eng.*, vol. 14, no. 2, p. 131, Dec. 2008, doi: 10.1007/s10664-008-9102-8.
[51] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: An Introduction*. Springer US, 2000.
[52] R. Kazman *et al.*, "A Case Study in Locating the Architectural Roots of Technical Debt," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, May 2015, vol. 2, pp. 179–188, doi: 10.1109/ICSE.2015.146.
[53] N. Zazworka *et al.*, "Comparing four approaches for technical debt identification," *Softw. Qual. J.*, vol. 22, no. 3, pp. 403–426, Sep. 2014, doi: 10.1007/s11219-013-9200-8.
[54] I. Griffith, D. Reimanis, C. Izurieta, Z. Codabux, A. Deo, and B. Williams, "The Correspondence Between Software Quality Models and Technical Debt Estimation Approaches," in *2014 Sixth International Workshop on Managing Technical Debt*, Sep. 2014, pp. 19–26, doi: 10.1109/MTD.2014.13.
[55] R. Marinescu, "Assessing technical debt by identifying design flaws in software systems," *IBM J. Res. Dev.*, vol. 56, no. 5, pp. 9:1-9:13, Sep. 2012, doi: 10.1147/JRD.2012.2204512.
[56] N. A. Ernst, S. Bellomo, I. Ozkaya, and R. L. Nord, "What to Fix? Distinguishing between Design and Non-design Rules in Automated Tools," in *2017 IEEE International Conference on Software Architecture (ICSA)*, Apr. 2017, pp. 165–168, doi: 10.1109/ICSA.2017.25.
[57] F. A. Fontana, R. Roveda, S. Vittori, A. Metelli, S. Saldarini, and F. Mazzei, "On Evaluating the Impact of the Refactoring of Architectural Problems on Software Quality," in *Proceedings of the Scientific Workshop Proceedings of XP2016*, New York, NY, USA, 2016, pp. 21:1–21:8, doi: 10.1145/2962695.2962716.
[58] F. A. Fontana, R. Roveda, and M. Zanoni, "Technical Debt Indexes Provided by Tools: A Preliminary Discussion," in *2016 IEEE 8th International Workshop on Managing Technical Debt (MTD)*, Oct. 2016, pp. 28–31, doi: 10.1109/MTD.2016.11.
[59] R. Baggen, J. P. Correia, K. Schill, and J. Visser, "Standardized code quality benchmarking for improving software maintainability," *Softw. Qual. J.*, vol. 20, no. 2, pp. 287–307, Jun. 2012, doi: 10.1007/s11219-011-9144-9.
[60] A. Mori *et al.*, "Evaluating Domain-Specific Metric Thresholds: An Empirical Study," in *2018 IEEE/ACM International Conference on Technical Debt (TechDebt)*, May 2018, pp. 41–50.

[61] L. Veado, G. Vale, E. Fernandes, and E. Figueiredo, "TDTool: Threshold Derivation Tool," in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, New York, NY, USA, 2016, pp. 24:1–24:5, doi: 10.1145/2915970.2916014.

[62] T. Döhmen, M. Bruntink, D. Ceolin, and J. Visser, "Towards a Benchmark for the Maintainability Evolution of Industrial Software Systems," in *2016 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement (IWSM-MENSURA)*, Oct. 2016, pp. 11–21, doi: 10.1109/IWSM-Mensura.2016.014.

[63] A. Yamashita, "Experiences from performing software quality evaluations via combining benchmark-based metrics analysis, software visualization, and expert assessment," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2015, pp. 421–428, doi: 10.1109/ICSM.2015.7332493.

[64] P. Oliveira, F. P. Lima, M. T. Valente, and A. Serebrenik, "RTTool: A Tool for Extracting Relative Thresholds for Source Code Metrics," in *2014 IEEE International Conference on Software Maintenance and Evolution*, Sep. 2014, pp. 629–632, doi: 10.1109/ICSME.2014.112.

[65] P. Oliveira, M. T. Valente, and F. P. Lima, "Extracting relative thresholds for source code metrics," in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, Feb. 2014, pp. 254–263, doi: 10.1109/CSMR-WCRE.2014.6747177.

[66] K. A. M. Ferreira, M. A. S. Bigonha, R. S. Bigonha, L. F. O. Mendes, and H. C. Almeida, "Identifying thresholds for object-oriented software metrics," *J. Syst. Softw.*, vol. 85, no. 2, pp. 244–257, Feb. 2012, doi: 10.1016/j.jss.2011.05.044.

[67] A. Mayr, R. Plösch, and C. Körner, "A Benchmarking-Based Model for Technical Debt Calculation," in *2014 14th International Conference on Quality Software*, Oct. 2014, pp. 305–314, doi: 10.1109/QSIC.2014.35.

[68] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data," in *2010 IEEE International Conference on Software Maintenance*, Sep. 2010, pp. 1–10, doi: 10.1109/ICSM.2010.5609747.

Chapter VIII. CONCLUSIONS AND FUTURE WORK

1. Conclusions and Contribution

The overall contribution of this dissertation comprises five (5) points; (a) it provides valuable undiscovered information on how open source web applications evolve over time since web-based systems had received limited attention in contrast to desktop ones, (b) considering the lack of empirical evidence on the relation between TD amount and TD Interest, it investigates to what extent the presence of TD in software modules slows down development pace by increasing the time and effort required for fixing bugs, (c) by acknowledging that it is the actual craftsmanship of the developers that causes the accumulation of TD, this work outlines the characteristics of the developers who tend to add TD in open source applications, (d) it sheds light into the reasons that drive developers to agree or disagree with automatically detected TD whose urgency is very often questionable by developers and (e) by acknowledging the lack of a basis regarding the detection and prioritization of TD among the existing TD tools, the current work proposes a methodology to extract a benchmark set of modules which are ranked as high-TD modules by three (3) TD tools altogether.

The following sub-sections develop the conclusions and the contribution of the conducted research in the context of the five goals of this dissertation. The goals were described, in summary, in the introductory Chapter I.

1.1. Evolution of Web Applications

The evolution of web applications relying on scripting languages such as PHP has received limited attention, despite the fact that PHP forms the basis upon which a huge number of web applications are developed. Driven by the wide adoption of PHP in web technologies, this dissertation investigates the evolution of 30 PHP web applications.

The main goal was to examine the validity of the eight laws of software evolution as stated by M. M. Lehman. These laws have been extensively studied in the context of software evolution for projects developed in compiled languages such as C and C++ and in a non-web related context.

The results confirm the validity of continuing growth and changes for the evolution of the examined PHP applications. However, for the examined projects the 2nd law on increasing complexity and the 8th law on the rapid decrease of the growth rate have not been confirmed. Although the root causes for this trend require further investigation it is reasonable to assume that this phenomenon could be attributed either to the programming language or to the practices in web application development.

The following implications of this part of dissertation can be identified.

With respect to software practitioners and managers:

- In the context of the investigation of Lehman's laws of evolution the employed measures can be used to assess the evolution of other products and examine whether any striking deviations from Lehman's observations are valid for their projects. Since most laws are not directly quantifiable, software maintainers could employ the same methodology with respect to the applied trend tests and indicators that have been analyzed for each law.

- Especially with respect to the evolution of quality vs. the increase of size contrasting the results for their own projects to those of the examined applications could highlight issues that warrant attention. For example, it should be regarded as a warning if their own PHP web projects do not success in allowing continuous changes combined with a non-increasing complexity, since this trend has been observed both for small and large open-source projects in this work. If, for example, a development team observes that complexity is constantly increasing, whereas large and complicated PHP systems manage to keep complexity stable or even reduce it over time, then, quality assurance should focus on ways to address the increasing complexity.
- The results suggesting that PHP web applications conform to a lifecycle model where continuous and steady development takes places (a finding confirmed by other studies as well), imply that development teams should opt for agile development practices, where constant change is embraced, rather than models assuming elaborate and preconceived specifications and planning.
- The results indicating that PHP web applications continuously change and grow, a finding shared by all other studies as well, imply that project managers should anticipate increased future needs for resources to maintain and sustain the existing systems.

With respect to software engineering researchers:

- Based on the findings indicating that PHP web applications do not suffer from software ageing, researchers can focus on the reasons that drive this improved behavior of PHP projects and investigate whether this is due to the language, the domain or the practices in web application development.
- Researchers are encouraged to investigate whether the same trends are valid for the evolution of systems written in other scripting languages so as to investigate whether similar maintenance patterns can be attributed to the nature of the employed languages (i.e. scripting vs. compiled).

Finally, for the specific group of research efforts that investigate the validity of Lehman's laws, empirical findings that suggest that: a) several laws are consistently not confirmed (e.g. Law VIII), or that b) some laws occasionally lead to inconclusive results (e.g. Laws IV and VII) or that c) some laws are quantified by divergent approaches (e.g. Law IV), imply that the rules might need to be examined in the context of contemporary software development and possibly be revisited.

1.2. Technical Debt and Corrective Maintenance

The results of this work suggest that TD amount is indeed correlated with maintenance effort. In particular, developers appear to spend more time on fixing issues in files with high levels of accrued Technical Debt, compared to files that present less TD. Therefore, project managers should take quality-oriented decisions to deter the appearance of software units with increased technical debt.

With respect to practitioners, the results provide additional evidence that TD undermines software maintenance and that it should be taken under consideration before any design and implementation decision. Moreover, the domain of the study suggests that TD appears to be important in a web context as well. Software engineers can take advantage of such empirical evidence to convince management about the importance and need to manage TD.

From a research perspective, since there is sufficient empirical evidence of the impact of TD amount on corrective maintenance, the need to devise a framework for assessing the associated risk and costs of managing TD becomes essential.

1.3. Personalized Assessment of Technical Debt Principal

Software development is a complex activity requiring experience, skills and significant mental effort. Artifacts produced by developers are systematically analyzed in terms of quality, which recently is successfully captured by the Technical Debt metaphor. This dissertation investigated, through a case study on four open-source PHP projects, the relation between introduced TD principal and developers.

The findings confirm the belief that developers' competencies vary, since the distribution of technical debt among developers is highly imbalanced. Moreover, different developers introduce different technical debt violations; however, some recurring violations can be identified across developers and projects.

Finally, there is no statistically significant evidence that more experienced developers introduce less technical debt per line of code. Such findings but more importantly the ability to perform a personalized assessment of technical debt can be a valuable tool for effective project management and self-assessment and improvement.

With respect to software project managers, resource allocation can benefit by assigning artifacts with increased technical debt interest probability to software engineers that tend to introduce less technical debt principal or even remove technical debt. In a similar line of thought, and without any intent to punish developers, managers could identify developers who impair software quality by introducing source code violations and technical debt instances and try to upgrade their coding habits, either by placing them next to more experienced developers or by calling them to reflect on their common violations. Appropriate guidelines or tooling to avoid the accumulation of particular violations can also be developed, based on the findings from previous projects.

With respect to software developers, the results on the personalized assessment of technical debt can be a valuable self-improvement tool. Developers can identify recurring problems that they consciously or unconsciously introduce as well as their locations in code. Moreover, critically analyzing their own performance with respect to TD against the rest members of their team can highlight opportunities for improvement.

1.4. Factors Affecting Decision to Repay Technical Debt

Existing software quality tools can yield extremely long lists of refactoring suggestions, deterring developers from adopting them. Thus, there is a need to determine which refactoring opportunities make sense for the developers depending on their background, nature and importance of the problem, surrounding code context, etc. This dissertation investigated various factors that potentially drive open-source software developers to accept or reject a suggestion to resolve a TD item.

According to results, developers appear to be largely influenced by the severity of a TD issue (i.e. Critical, Major, Minor and Info as no Blocking issues were identified). For example, it is 21.5 times more probable that a Critical issue will be classified as needing resolution compared to an Info issue. This finding is reasonable, as a Critical code issue like "*String literals should not be duplicated*" is perceived as more urgent to be resolved than an Info code issue like "*Comments should not be located at the end of lines of code*".

The broader characterization of the TD issue also seems to have an effect on the developer's decision. For example, if an issue pertains to Testability (like "*Expressions should not be too complex*") it is 3.9 times more probable to be considered as needing resolution than an issue related to Maintainability (like "*Sections of code should not be "commented out"*").

Finally, developers do not tend to accept suggestions for revising their own code: it is 3 times more likely that a developer who has not participated in a project agrees with a suggestion to remove a TD issue, than a developer who is a contributor. This might be related to the particular practices within the community of a software project where certain violations are not considered as harmful because the evolution of the project might have been unaffected by their presence.

On the other hand, developers' decisions appear to be unaffected by factors such as the frequency of modifications to the file under study (reflected in the Files Modifications Ranking variable), the time required to fix an issue and the total TD in the examined file.

These findings can be valuable to researchers and practitioners by guiding the design of more efficient tools that suggest refactorings with a higher probability of being adopted by the developers.

1.5. Benchmark of Technical Debt Liabilities

The Technical Debt metaphor successfully captures, in monetary terms, the penalty that has to be paid because of shortcuts during software development. These shortcuts are known to introduce architectural, design and code inefficiencies in software systems and various TD tools aim at identifying them by testing the source code against specific rulesets. However, TD tools provide different estimates of TD principal pointing to different mitigation actions. These discrepancies make a lot of people in academia and practice skeptical about the validity of existing TD tools and hinder the further development of TD research as no ground truth for accurate TD instances can be established.

To address these limitations in the TD community an empirical study was performed whose goal was twofold: (a) to determine the level of agreement among three well-known TD tools and (b) build agreement-based benchmarks of high-TD classes/files from a dataset resulting from fifty (50) open-source projects. Inter-rater agreement has been assessed, using Kendall's W coefficient of concordance. To capture the diversity of the examined tools with the aim of identifying representative class profiles, archetypal analysis was conducted. Once the derived reference assessments are characterized, it is straightforward to extract sets of classes exhibiting similarity to a selected profile (e.g. that of high TD levels in all employed tools) and in this way establish a basis.

The findings of the inter-rater agreement analysis suggest that there is a statistically significant and strong agreement among the three TD tools on the measurement of TD at class level. However, a substantial degree of disagreement has also been observed for the measured TD level for numerous classes. The application of the archetypal analysis revealed that three types of reference assessments can successfully capture the spectrum of TD measurements provided by three tools: One set of archetypes represents classes identified as high-TD modules by only one of the tools, the second profile encompasses classes for which two of the tools agree on the measured TD level, while the final type of archetype signifies a high amount (or low amount) of TD based on the results of all applied tools. Selecting the classes in the vicinity of the latter archetype yields an agreement-based benchmark of classes tagged as high-TD by all tools. Such benchmarks, beyond their value as fields of study for poor development practices that led to low quality classes, can potentially form the basis for training more sophisticated TD identification and measurement approaches.

Below, the main outcomes of this part of dissertation are discussed, from the perspectives of practitioners and researchers.

Overcome construct validity threats in research (researchers)

As mentioned in the introductory section, the research community within the TD field lacks an ultimate process to accurately capture TD principal and thus, any empirical study or technique based on TD estimates runs the risk of not accurately measuring the real-world phenomenon under study. Each tool follows its own approach for detecting and measuring TD, based on a distinct ruleset, yielding a different amount for the total TD, but also pointing to different parts of the code that need to be mitigated, compared to other tools. There are several studies trying to identify high-TD modules and studies investigating the association of accrued TD with other factors. However, such approaches are heavily dependent on the employed tool for suggesting the ground truth, that is, the modules that actually have TD liabilities and need to be fixed. Apparently, because each tool evaluates TD in a different way, the generalizability of these approaches is threatened to a large extent.

The two aspects of the proposed methodology, that is, the estimation of inter-rater agreement among TD tools and the use of archetypal analysis for identifying classes having a desired profile (e.g. high-TD levels by all tools) can be applied by researchers to form a more reliable basis for their experiments. More conveniently, researchers can also employ the already available benchmarks of high-TD classes (but also classes having a different profile if needed) from the online TD Benchmarker web application. Consequently, leveraging the power of multiple TD tools using the proposed approach can assist in the mitigation of construct validity threats that is currently present in the field of TD.

Highlight critical modules with validated highest TD (practitioners)

Despite the widespread adoption of the TD metaphor, it is far from clear which tool IT managers should integrate in the development and maintenance process. Employing more than one TD tool for the evaluation of their software might be a costly option, since most of the existing tools are available only with a commercial license. Moreover, each tool requires significant effort to deploy, properly configure and familiarize with. However, even if a development team employs more than one tool, the union of all findings, would result in an unrealistic amount of suggestions, rendering the process intractable. Based on the proposed methodology, practitioners can highlight the classes that have been identified as high-TD classes by all employed tools leading to a manageable number of target classes. Development teams can take advantage of such agreement-based benchmark sets and focus only on the modules of their system that are validated as high-TD modules. With respect to the benefit of the already derived benchmarks from the analyzed systems, developers can focus on the classes close to the Max-Ruler archetype and gain insight into the root causes of the accumulation of TD in these classes and potentially avoid non-ideal coding practices in the future. Moreover, the non-unanimous archetypes (Rebel and Partner archetypes) can be valuable, as well. The existence of these archetypes is the key factor that differentiates one tool from the others. If only unanimous archetypes existed, this would mean that all tools generate the same results pointing to the same classes/files with accrued TD principal. Through the exploration of classes/files in the vicinity of non-unanimous archetypes development teams can gain insight into how TD tools differ on the measurement and prioritization of TD principal. With such knowledge, developers can more confidently invest in the TD tool that best fits their perception of when a class/file is tagged as high-TD (or low-TD).

Collection of available TD tools (researchers and practitioners)

Last but not least, another contribution of the current work is the localization and collection of available TD assessment tools, as presented in Section 2. The list is by no means an exhaustive one, as numerous other tools offer functionality related to the identification of code smells, anti-patterns, rule violations, excessive metric values, etc. all of which are indicators of the existence of TD in software. Nevertheless, the presented tools can serve as starting point both for practitioners who are searching for a TD tool to

integrate into their development process as well as researchers who are seeking an appropriate assessor of TD principal. In both cases, the proposed methodology can assist in the critical appraisal of the agreement or the diversity among tool findings.

2. Future Work

Existing TD tools generate large lists of detected TD violations which can be overwhelmingly long for large projects. Many of the detected violations are considered as non-important by the developers or they are even ignored. In an attempt to bring to the surface only violations that are indeed urgent to fix, in this dissertation a benchmark set of high-TD classes was extracted, derived from the results of three major TD tools. An extension of this work would be to enforce the credibility of the benchmark set with the inclusion of more TD tools and more analyzed projects. This way, developer teams can be more confident on the prioritization TD management. Moreover, the interpretation of the results considering different perspectives, such as development context, role of developers (tester, designer, analyzer, etc.) was beyond the scope of the current research. Nevertheless, this forms a really interesting area of future work. Another interesting line of research would be to investigate to which degree TD tools are compliant with the guidelines of the OMG Specification on Automated Technical Debt Measure. Furthermore, the nature of the examined rules by each tool might be a decisive factor for the TD principal estimates per class/file. Drilling down to the level of individual rule violations which are detected by each tool, can shed light into the cause of their agreement or discrepancy. Another valuable line of future work would be to investigate the similarity among the examined rules by mapping the rules adopted by each tool to the rules employed by the other tools.

It is a common ground that web technology has boomed over the last five years with the advance in cloud computing and containerization. As a result, more languages and technologies have gained a respected share in the global pie of web content. Modern web applications adopt the microservices design where each microservice can be written in any language. To this end, a valuable future work would be to expand the study on the evolution of software quality and technical debt to web applications that combine different languages and technologies. Another interesting line of further research would be to compare the evolution of such multi-paradigm web applications against that of "conventional" desktop systems, in order to investigate whether there are differences in the trends of quality and TD. Such evidence would be helpful in determining whether development practices for web applications adhere to the principles of building large-scale, multi-person, multi-version software systems or whether the benefits is the result of their architecture, which is often strictly dictated by the platforms being used.

The results of the current research regarding the Lehman's Laws of software evolution do not provide clear evidence that open source web applications suffer from the so-called phenomenon of software ageing. The deeper investigation on the factors that prevent the accumulation of TD in some systems can be a valuable research area. For example, which is the best practice to manage TD? Perform mitigation actions (i.e., refactor, rewrite code) or take proactive measures to prevent the introduction of TD in the first place (e.g. by integrating tools with IDEs and develop code step by step)? Taking also into account the cost of each TD management approach could lead to an exploration of the tradeoffs between software quality improvement and the required effort. Many of the existing approaches, simply assume that achieving a non-optimal software quality (i.e. not repaying the principal of TD) will result in increased maintenance effort (as captured by TD interest). However, one should also consider that any savings from not addressing TD principal, and especially in large corporation, might have been directed to other sorts of investments, like the development of additional features or produces, or even to conventional financial investments.

Finally, acknowledging that it is the actual craftsmanship of the developers that cause the accumulation of TD, the current dissertation investigated the relation between developers' characteristics and their tendency to introduce code inefficiencies. The outcomes suggest that a personalized assessment of TD can be a meaningful research direction that unveils interesting relations that can guide Technical Debt Management. Therefore, the topic deserves further investigation. Some tentative future research direction would be a personalized assessment of TD interest, a detailed analysis of specific violations with respect to their criticality, and an elaborate personality characteristics model that will provide a more accurate profile of TD-prone developers.

Publications

The work of this dissertation has been documented in a number of papers that have been published in International Conferences and Journals.

1. Journals

J1. Amanatidis, Theodoros & Chatzigeorgiou, Alexander & Ampatzoglou, Apostolos. (2017). The Relation between Technical Debt and Corrective Maintenance in PHP Web Applications. Information and Software Technology. 87. 10.1016/j.infsof.2017.05.004.

J2. Amanatidis, Theodoros & Chatzigeorgiou, Alexander. (2016). Studying the Evolution of PHP Web Applications. Information and Software Technology. 72. 48-67. 10.1016/j.infsof.2015.11.009.

Submitted for publication and revised (awaiting final response)

J3. Amanatidis, Theodoros & Mittas, Nikolaos & Moschou, Athanasia & Chatzigeorgiou, Alexander & Ampatzoglou, Apostolos & Angelis, Lefteris. (2020). Evaluating the Agreement among Technical Debt Measurement Tools: Building an Empirical Benchmark of Technical Debt Liabilities. Submitted for publication to the Empirical Software Engineering Journal (EMSE).

2. Conferences

C1. Amanatidis, Theodoros & Mittas, Nikolaos & Chatzigeorgiou, Alexander & Ampatzoglou, Apostolos & Angelis, Lefteris. (2018). The developer's dilemma: Factors affecting the Decision to Repay Code Debt. Proceedings of the 2018 International Conference on Technical Debt (TechDEBT), Gothenburg, Sweden. 10.1145/3194164.3194174.

C2. Amanatidis, Theodoros & Chatzigeorgiou, Alexander & Ampatzoglou, Apostolos & Stamelos, Ioannis. (2017). Who is Producing More Technical Debt? A Personalized Assessment of TD Principal. Proceedings of the 9th International Workshop on Managing Technical Debt (MTD' 17), Cologne, Germany 10.1145/3120459.3120464.

C3. Chatzigeorgiou, Alexander & Ampatzoglou, Apostolos & Ampatzoglou, Areti & Amanatidis, Theodoros. (2015). Estimating the Breaking Point for Technical Debt. Proceedings of the 7th International Workshop on Managing Technical Debt (MTD' 15), Bremen, Germany 10.1109/MTD.2015.7332625.

References

To facilitate the tracking of the bibliography by the reader, all references (IEEE Style) are listed in the end of each chapter.