Who is Producing More Technical Debt? A Personalized Assessment of TD Principal

Conference Paper · May 2017

CITATIONS
0

READS
9

4 authors, including:

Alexander Chatzigeorgiou
University of Macedonia
160 PUBLICATIONS **1,498** CITATIONS
SEE PROFILE

Apostolos Ampatzoglou
University of Groningen
53 PUBLICATIONS **247** CITATIONS
SEE PROFILE

Ioannis Stamelos
Aristotle University of Thessaloniki
186 PUBLICATIONS **2,362** CITATIONS
SEE PROFILE

Who is Producing More Technical Debt? A Personalized Assessment of TD Principal

Theodoros Amanatidis
Department of Applied Informatics
University of Macedonia
Thessaloniki, Greece
tamanatidis@uom.edu.gr

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

Apostolos Ampatzoglou
Department of Computer Science
Aristotle University of Thessaloniki
Thessaloniki, Greece
apamp@csd.auth.gr

Ioannis Stamelos
Department of Computer Science
Aristotle University of Thessaloniki
Thessaloniki, Greece
stamelos@csd.auth.gr

ABSTRACT

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 costs, efficient software project management cannot occur without recognizing the relation between developer characteristics and the tendency to evoke violations that lead to TD. In this paper, we investigate 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 in the study.

CCS CONCEPTS

• **Software and its engineering** → Software creation and management → Software post-development issues→ *Maintaining software*• **Social and professional topics**→ Management of computing and information systems → Software management → *Software maintenance*

KEYWORDS

Technical Debt; Software Maintenance; Project Management

MTD 2017, May 22, 2017, Cologne, Germany
 ISBN 978-1-4503-4486-9/17/04…$15.00

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, in this paper we present 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, we investigate 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. In the context of this study 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. We note 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 paper 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, we conclude in Section 7.

2 RELATED WORK

In this section we present 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 amount 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.

Our study 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 study, 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: 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. The first research question is formulated as follows:

RQ$_1$: *Is TD uniformly distributed among 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. The second question is formulated as:

RQ$_2$: *Which TD violations are introduced by the developers of 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. The last question is formulated as:

RQ$_3$: *What is the relation between TD and the maturity of developers in a software project?*

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 our study, we employed 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. 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 I. ***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 study (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 I: 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) we recorded the following variables in order to answer the research questions that have been set:

[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 we recorded the number of lines that have been modified during each commit (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 we obtained the most recent year's commit data for every examined project 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[1]. 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, we checked out the source code corresponding to each commit 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.

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

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[2] 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 we 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.

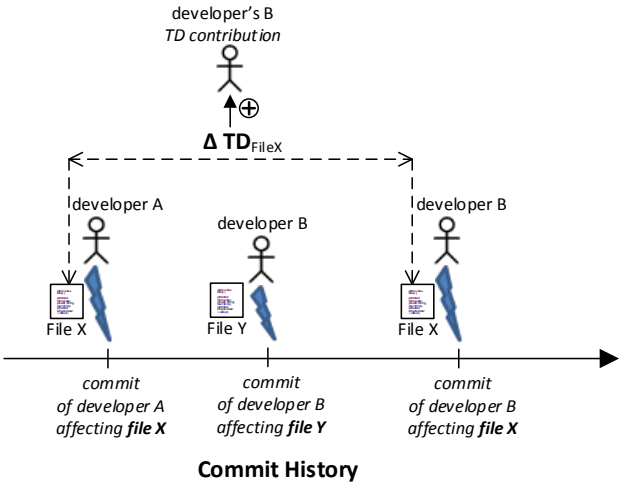


Figure 1: Process of obtaining TD deltas for each developer

3.4 Data analysis

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

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

[2] 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

To investigate whether developers have a tendency to introduce particular TD violations (RQ_2) we used a heatmap. 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 we display the findings as scatterplots (developer age vs. normalized TD) and test the hypothesis whether normalized TD depends on age 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 we have performed an independent study t-test, by differentiating between less- and more-experienced developers (we used as threshold the age in days corresponding to 50% of the longest experience). The analysis strategy per research question is summarized in Table II.

Table II. 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

In this section we present the results of the study organized per research question along with an interpretation of the findings.

4.1.1 Distribution of TD among Developers

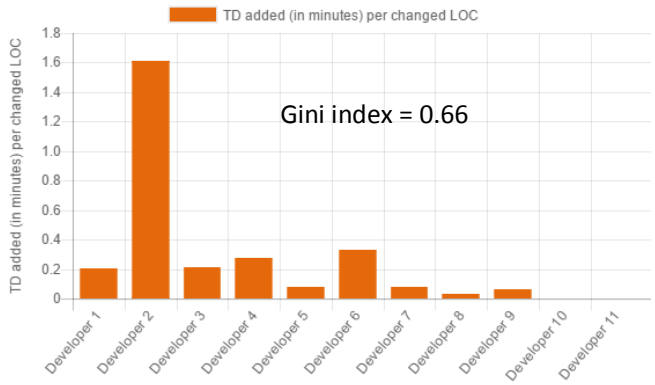
Figure 2 illustrates the distribution of the contributed TD during the examined time frame among the developers who performed commits in each project. To avoid biasing the results by the

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.

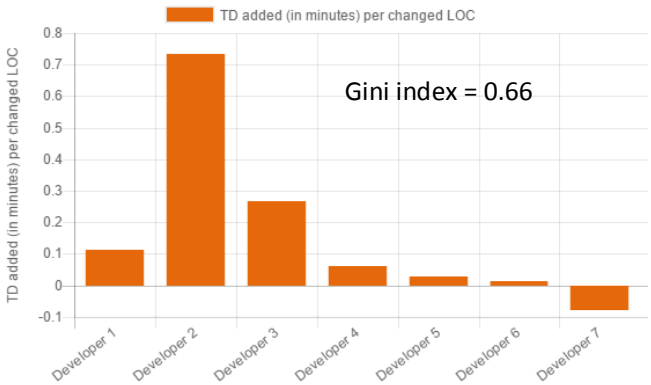
The pattern observed in each plot presents similarities across projects. A limited number of developers (e.g. *Developer-2 for Laravel* and *Developer-5 and Developer-11 for CakePHP*) contribute a significant portion of the system's technical debt (in terms of TD per line of code), while the majority 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.

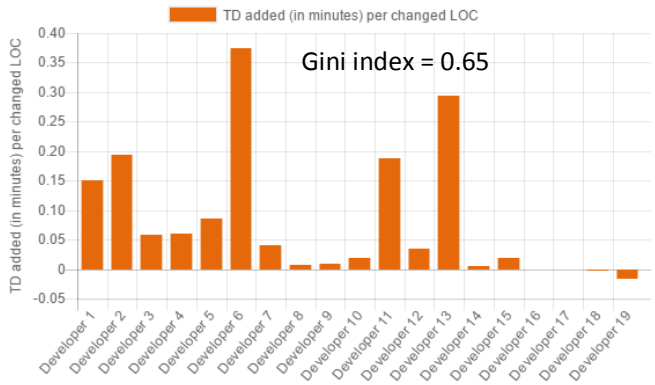
We claim that TD principal is not equally distributed across developers since at least one of them stands up as a main source of producing violations (and therefore introducing principal). On the contrary, there are cases in which developers consistently remove violations (i.e., repay TD). However, this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



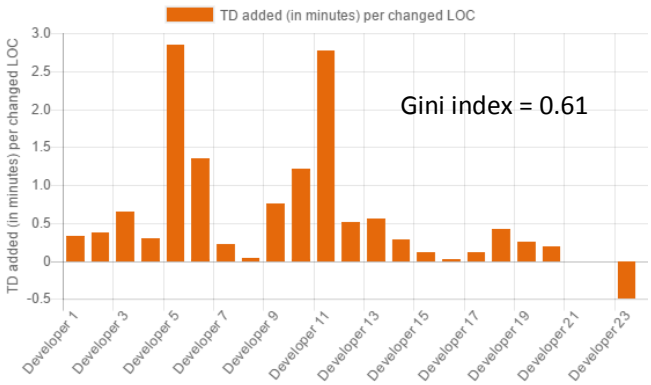
(a) Laravel (core)



(b) Composer



(c) Yii2



(d) CakePhP

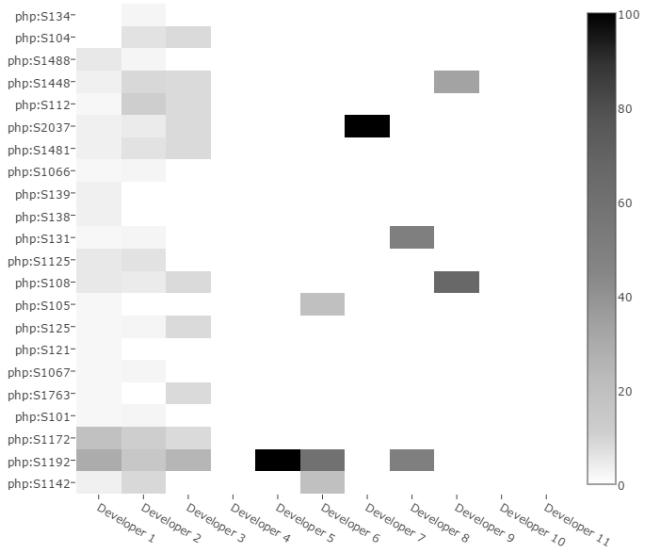
Figure 2: Distribution of TD among developers

4.1.2 TD Violations per Developer

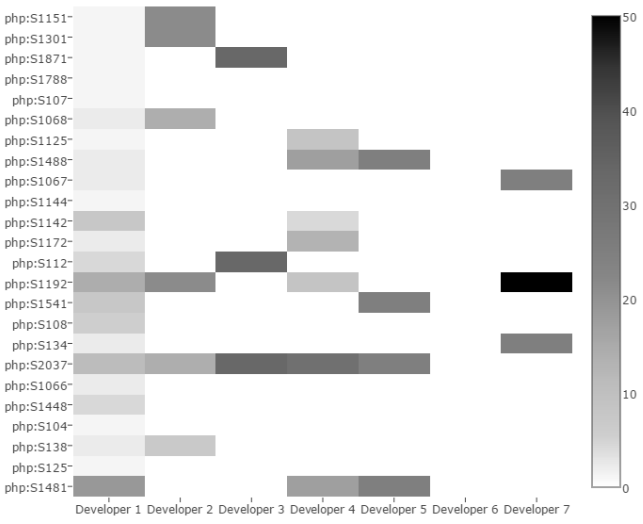
Figure 3 illustrates the most common violations in each of the examined projects against the developers who introduce them, in the form of a heatmap. The darker the color the more violations of the corresponding type are introduced by the indicated developer. A row that is relatively dark across all 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.

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

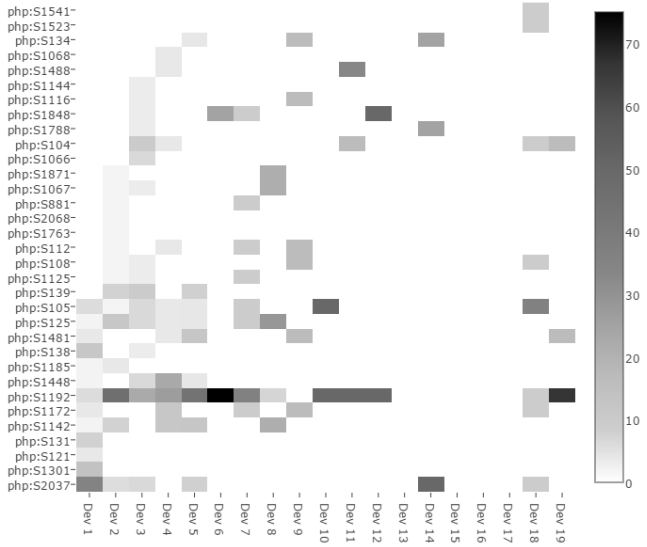
cells indicate common violation types introduced by many developers. Such a violation is violation '`php:S1192`' (of critical importance) in all projects. According to SonarQube this violation indicates the presence of String literals which are duplicated, rendering the process of updating all occurrences 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.



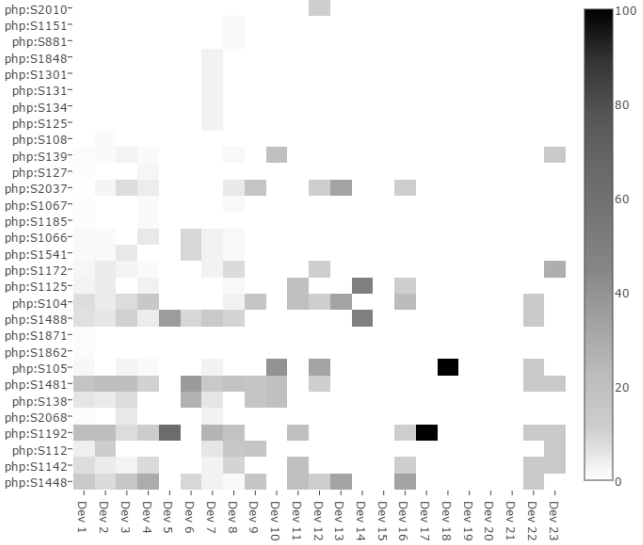
(a) Laravel (core)



(b) Composer



(c) Yii2



(d) CakePhP

Figure 3: TD violation types per developer

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, we have pointed out to specific frequently recurring violations for: (a) the same project, (b) the same developer, and (c) across all projects.

4.1.3 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).

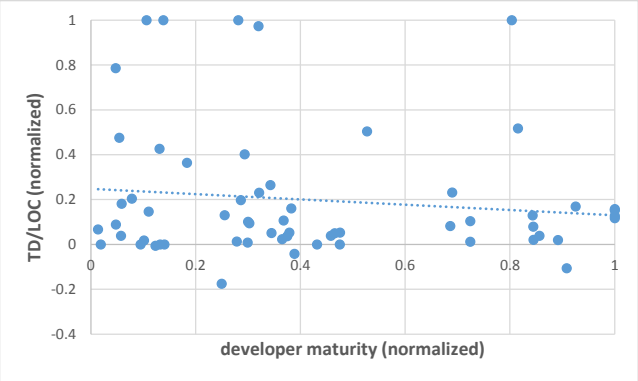


Figure 4: Introduced TD versus developer maturity

To further investigate whether developer's maturity plays any role in the amount of introduced TD principal we have performed an independent study t-test. However, the results of the test have not suggested the rejection of the null hypothesis (sig: 0.8). Therefore, we 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 we can observe that a larger number of immature developers is concentrated in the top-20% most TD-incurring developers (5 immature against 1 experienced). This finding, in conjunction with the declining trendline in the scatterplot opens up and 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

In this section we present and discuss 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 we explicitly report the design of the case study and the statistical tests that have been performed.

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 our possibility to generalize the findings. To mitigate this threat we have selected four widely known PHP projects 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 paper, we have attempted to investigate, 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.

ACKNOWLEDGEMENT

This work was financially supported by the action "Strengthening Human Resources Research Potential via Doctorate Research" of the Operational Programme: "Human Resources Development Program, Education and Lifelong Learning, 2014-2020", implemented from State Scholarship Foundation (IKY) and co-financed by the European Social Fund and the Greek public (National Strategic Reference Framework (NSRF) 2014 – 2020)

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.