Noname manuscript No.
(will be inserted by the editor)

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

George Digkas · Apostolos Ampatzoglou* · Alexander Chatzigeorgiou · Paris Avgeriou

Received: date / Accepted: date

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

Work reported in this paper has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 801015 (project EXA2PRO) regarding the contribution of A. Chatzigeorgiou and A. Ampatzoglou, as well as from ITEA3 and RVO under grant agreement No. 17038 (VISDOM), regarding the contribution of P. Avgeriou.

G. Digkas, P. Avgeriou
Institute of Mathematics and Computer Science, University of Groningen, Netherlands
E-mail: g.digkas@rug.nl, paris@cs.rug.nl

G. Digkas, A. Ampatzoglou, A. Chatzigeorgiou
Department of Applied Informatics, University of Macedonia, Greece
E-mail: g.digkas@uom.edu.gr, a.ampatzoglou@uom.edu.gr, achat@uom.edu.gr
* Corresponding Author: Apostolos Ampatzoglou a.ampatzoglou@uom.edu.gr

Keywords TD temporality · case study · new code debt · metrics fluctuation.

1 Introduction

Technical Debt (TD) at the code level refers to inefficiencies introduced in the source code of an application during the implementation or the maintenance phase [20]. These inefficiencies manifest themselves as violations of coding standards, complex and hard to understand code, code duplicates, etc. [19]. According to Alves et al. [2] code technical debt is the most studied type of technical debt, and based on Ampatzoglou et al. [4] it is one of the most important in industry.

One of the certainties in software development is continuous change [18], and code technical debt is no exception to this. In fact, there has been significant work on how code technical debt evolves and how it accumulates over time. However, existing studies have looked at technical debt evolution as a whole, without distinguishing between technical debt that is added as new code, and technical debt incurred through the modification of existing code. In this paper, we focus only on the introduction of new code technical debt, i.e., TD inserted in the system in the form of new Technical Debt issues. More specifically we study new methods (our scope is object-oriented systems) that contain technical debt and we look at the introduction of new technical debt as a temporal phenomenon. This can shed light on whether technical debt is introduced in an almost flat rate, whether large volumes of technical debt are introduced in infrequent timestamps, or whether large volumes of TD are frequently introduced along evolution; examples of these three cases are presented in Figure 1. Consequently, in the second and third case, we need to revise our technical debt management techniques to focus on those times where large TD is introduced to prevent or limit the phenomenon. Broadly speaking, the introduction of TD items in new code can either depend on the capabilities of the development team or be associated with external factors such as feature requests in short time, lack of sufficient time for testing, etc. The former would be reflected in a uniform introduction of TD along evolution, whereas the latter would be recognized by fluctuations in the introduction of TD. Knowing the particular circumstances can help towards self-improvement of the development process so as to address the root causes of TD accumulation.

Focusing on technical debt that is introduced by new code, as opposed to technical debt that is introduced by modifying existing code, can provide a unique insight. Specifically, the new Technical Debt issues introduced by new methods at each commit (either new methods in existing classes or new methods in entirely new classes) reflect more accurately the type of problems and the timepoint at which they are introduced. In other words, new methods are more representative of the developers' practices and knowledge level, compared to method modifications whose type and timeliness is often dictated by the need to fix a bug or to extend an already existing functionality. Thus, we study the temporality of technical debt through a clearer source. Furthermore, in our previous work we established that through the systematic introduction of 'clean' new code, a software system can gradually be freed of its technical debt [10].

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

To achieve this goal, we explore the evolution of 47 open source projects from two well-known ecosystems, namely: Apache Software Foundation (ASF) and Eclipse Foundation (EF). However, since ecosystems exhibit particular habits and regulations, we have included additional OSS projects from GitHub to increase the generalizability of the findings. In particular, we track the number of new Technical Debt issues inserted in each commit. Next, we create a 6-month sliding window (the duration of the time window has been set to 6 months as in the study by Hassan on predicting faults using the complexity of code changes [15]), and we calculate the cumulative number of inserted Technical Debt issues for each window, as well as the number of commits in the same window and the weighted (by the contribution of each developer) average experience of the development team. To answer the first question (Q1), we use a metric property (termed SMF—see Section 3) that is able to assess metrics' fluctuation along time and characterize them as either stable

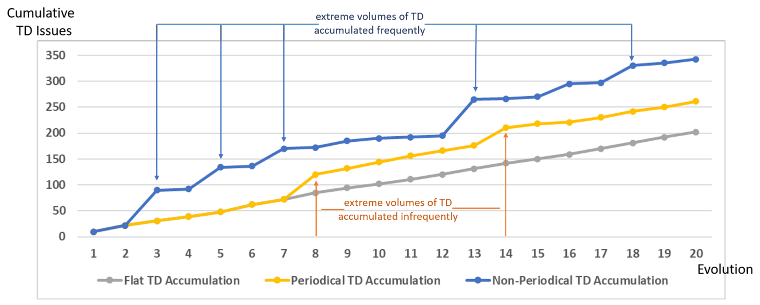


Fig. 1: TD Introduction Temporality

or sensitive. To answer the second question (Q2) we correlate the number of inserted Technical Debt issues with a) the number of commits for each window and b) the weighted experience of the development team. The reporting and interpretation of the results is performed at the project level.

This study is an extended and revised version of our previous work [9] in which we also explored the temporality of Technical Debt issues introduction. The main differences of this work compared to the previous one, are as follows:

- **Sample size**. It explores a significantly larger dataset of 47 instead of 27 projects, resulting into the exploration of (approx.) 2.5K instead of (approx.) 2K time windows.
- **Generalizability**. It strengthens the generalizability of our results, since it performs the analysis on two software ecosystems (Apache and Eclipse projects), compared to only one. In addition, we now explore isolated projects, i.e., not related to a specific ecosystem.
- **Accuracy of data analysis**. The analysis of spikes along evolution (i.e., the timestamps with large introductions of technical debt amounts) is performed through statistical analysis (extreme values detection), instead of visual observation.
- **Exploration of additional evolution parameters**. We explore also the relation between developers' experience and Technical Debt issues introduction through an additional research question. In the original paper, we had only looked into the relation between Technical Debt issues introduction and development effort intensity.

The rest of the paper is organized as follows: in Section 2 we present related work and in Section 3 background information important for understanding the study. In Section 4, we present the design of the case study, while Section 5 elaborates on the results. Section 6 interprets the results and provides implications for researchers and practitioners. Finally, in Section 7 we present threats to validity and in Section 8, we conclude the paper.

2 Related Work

Since this paper focuses on the introduction of technical debt over time, we organize this sub-section into causes of technical debt introduction (see Section 2.1) and technical debt evolution (see Section 2.2).

2.1 Causes of Technical Debt Introduction

Quite recently the research consortium of the InsighTD project[1] has explored the causes and effects of technical debt accumulation in various countries along the globe (e.g., Brazil, Chile, Colombia, Serbia, USA, etc.) through a family of surveys [23,25,26]. Through these research efforts they have identified in

[1] `http://www.td-survey.com/`

total 78 causes and 66 effects. Among the identified causes of technical debt, close deadline, inappropriate planning, and lack of knowledge/experience have proven to be the most prolific ones. Additionally, Tufano et al. [29] studied the evolution of code smells with the goal of understanding when and why code smells are introduced and observed the life cycle of five code smells. The results indicate that: (a) in the majority of the cases code smells are introduced with the creation of the corresponding classes or files; (b) while projects evolve, "smelly" code artifacts tend to become more problematic; (c) new code smells are introduced when software engineers implement new features or when they extend the functionality of the existing ones; (d) the developers who introduce new code smells, are the ones who work under pressure and not necessarily the newcomers; and (e) the majority of the smells are not removed during the project's evolution and few of them are removed as a direct consequence of refactoring operations.

According to Kazman et al. [16] who conducted a case study on the roots of architecture debt (ATD), is extremely common and probably the most important type of technical debt because it consumes the largest percentage of maintenance effort. Their findings suggest that architectural debt is extremely easy to introduce: programmers typically want to introduce new features or fix bugs; however, by changing the code they often undermine the architectural structure leading to the accumulation of ATD. Finally, Martini et al. [21] conducted a case study on five software companies to understand the causes that introduce ATD. Large software companies try to deliver as fast as possible in order to satisfy their customers' needs, usually taking shortcuts, thereby introducing ATD. If the debt is not paid-off, it starts to accumulate and this makes feature development more difficult. However, we clarify that the current study does not deal with ATD but rather focuses on code-level TD.

2.2 Evolution of Technical Debt

Although technical debt is a multifaceted concept, one of the key constituents of code technical debt is the presence of code smells. One of the first studies that investigate the evolution of code smells was conducted by Olbrich et al. [22]. They investigated the evolution of two code smells, God Class and Shotgun Surgery, on two OSS projects. The results show that along software development, there are phases where the number of code smells can either increase or decrease and those phases are not affected by the size of the systems. Chatzigeorgiou and Manakos [7] have investigated the evolution of the Long Method, Feature Envy, State Checking, and God Class smells in two open-source software projects. The results suggested that as projects evolve the number of smells tends to increase. Another interesting finding is that a significant percentage of smells was not due to software ageing, since some smells were present right from the first version of the code in which they reside. Peters and Zaidman [24] studied the lifespan of the God Class, Feature Envy, Data Class, Message Chain Class, and Long Parameter List smells. The analysis of eight open-source software projects, confirmed that the number of smells increases, as projects evolve.

Digkas et al. [11] tracked the evolution of technical debt in sixty-six open-source Java projects by the Apache Software Foundation, over a period of 5 years. In order to detect issues that incur technical debt, they relied on SonarQube. The results show that on the one hand, there is a significant increasing trend on the size, complexity, number of Technical Debt issues, and the total technical debt over time, which seems to confirm the software aging phenomenon. But on the other hand, when technical debt is normalized over the non-commented lines of code, an evident decreasing trend over time is present for many of the projects. This could possibly be attributed to: (a) developers that perform refactoring activities and fix some of the open Technical Debt issues; or (b) developers that introduce better quality code in each commit (compared to the project's existing code base). We summarize the key findings of the related work in Table 1.

Table 1: Causes and Evolutionary Patterns of TD/smells

Causes	Study
Close deadlines, inappropriate planning, lack of knowledge/experience	Pérez et al. [23], Ramač et al. [25], Rios et al. [26]
Pressure	Tufano et al. [29]
Architecture flaws	Kazman et al. [16]
Pressure to deliver	Martini et al. [21]
Evolutionary Patterns	**Study**
Smells tend to increase and decrease in different phases	Olbrich et al. [22]
Smells increase monotonically	Chatzigeorgiou and Manakos [7]
Smells increase monotonically	Peters & Zaidman [24]
TD increases monotonically (Normalized TD decreases for some projects)	Digkas et al. [11]

3 Background Information

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

Identifying Technical Debt issues on new code. SonarQube is one of the most widely used tools for assessing the level of code technical debt present in a software system [3,10]. According to a recent overview of TD tools [6] SonarQube is by far the most popular tool based on its trace in the scientific literature and online media channels. We have relied on version 7.9.2 to identify and quantify technical debt in individual commits throughout the history of the examined projects. SonarQube estimates technical debt principal using a set of predefined rules[2] such as *Methods should not be too complex*, *Inheritance tree of classes should not be too deep*, or *Classes should not be coupled to too many other classes*. Each identified issue is assigned a remediation cost corresponding to the time required to fix the corresponding rule violation. The sum of the remediation costs for all issues yields the reported TD principal.

As *new code* we consider the new methods which are introduced in each commit. Such methods can be added either in existing classes or in entirely new classes. We do not consider new code in the form of new instructions in existing methods (i.e. modifying methods). The reason is that, beyond the complexity of tracking changes at the instruction level, an entirely new method conveys better the programming habits of a developer as it refers to a complete, self-standing piece of functionality. To distinguish the newly inserted methods for each commit from the deleted, modified, renamed, and unchanged methods, we rely on the Gumtree Spoon AST Diff tool [13]. Our approach identifies all changes per commit at the file-level, i.e. we detect files which have been added, modified, renamed, and deleted. For the added files or classes, we consider all of their methods as new. For the modified and renamed files we compare their representation in the form of an Abstract Syntax Tree with the one of the previous revision; subsequently we identify the newly inserted methods in existing classes.

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

Assessing the fluctuation of Technical Debt issues. Observing the evolution of a metric value throughout the history of a software project can be considered as the analysis of a time series. The time series can exhibit volatility depending on the metric itself and on the nature of changes throughout the history. In previous work, we have defined Software Metrics Fluctuation (SMF) as "*the degree to which a metric score changes from one version of the system to the other*" [5]. According to the SMF property a metrics can be characterized as ***sensitive*** (changes induce high variation on the metric score) or ***stable*** (changes induce low variation).

As this study focuses on the evolution of the quality of new code, in terms of its TD, we make use of the SMF property. In particular we employ the `mf` measure [5] which is defined as: "*the average deviation from zero of the difference ratios between every pair of successive versions*". We adapt the `mf` measure for the case of Technical Debt issues, resulting in formula (1). *TDissues(i)* refers to the number of Technical Debt issues identified at version i of the history, while n is the total number of analyzed time snapshots (e.g. versions). A zero deviation of the number of Technical Debt issues from that

[2] `https://rules.sonarsource.com/`

of the previous version implies no fluctuation. The squared root of the second power of the ratio of the difference between any two successive versions yields a measure similar to the standard deviation.The closer to zero mf gets, the more stable the number of Technical Debt issues is.

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

4 Case Study Design

Case study is an observational method that is used for studying a phenomenon in its real-life context. In our study, the phenomenon is the temporality of TD in new code, while the context refers to the evolution of open-source software projects. In this section, we present the design of the case study, organized based on the linear-analytic structure as described by Runeson et al. [27].

4.1 Research Questions

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

RQ$_1$: *Does the number of Technical Debt Issues introduced by new code fluctuate along evolution?*

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

RQ$_2$: *Does the amount of Technical Debt introduced by new code correlate to the activity or the experience of the developers?*

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

4.2 Cases and Units of Analysis

This study is characterized as a multiple, embedded case study [27], in which the cases are open-source software (OSS) projects, while the units of analysis are the source code commits (per project) over different time periods. Specifically, for each project, we analyse the number of code Technical Debt issues added over 6-month time periods across the project history (see Section 4.3 for more details). The reason for selecting to perform this study on open-source software systems is the vast amount of data that is available in terms of revisions and classes. The long history that is available for each project enables researchers to observe overall trends in the evolution of their quality. To retrieve data from high-quality projects that evolve over a period of time, we looked into ASF and EF projects [12] as well as additional OSS projects and investigated the projects presented in Table 2. The demographics of Table 2 include the number of classes, number of non-commented lines of code, the number of analyzed revisions, the date of the first commit, number of issues in the corresponding GitHub issue tracker as well as the number of developers per project. The two ecosystems have been selected since they are well-known in the software engineering community for their quality, and structured development processes. On the one hand, Eclipse is an industry-driven initiative involving around 100 companies, universities, and contributors who deliver OSS based on the Eclipse environment. On the other hand, Apache Software Foundation is a highly successful initiative, made up of individuals, that provides popular, high quality, OSS–providing support for over half of the world's websites. Both communities follow a mature approach in developing software, having established processes to face the problems inherent to software development. However, to reduce generalizability threats from the study of projects belonging to well known and systematically maintained ecosystems, we considered twelve additional OSS projects.
The selection of projects was based on the following criteria:

– The software is actively maintained. To ensure this, we sorted projects based on the date of their last commit.
– The software is written in Java and uses Maven as a build tool. This ensures that the project can be built and allows the retrieval of the project's language version from the corresponding pom.xml file.

Table 2: Selected Projects' Demographics

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

- The software contains more than 90 classes to ensure the inclusion of systems with a substantial size, functionality and complexity.
- The software has more than 1000 commits. This criterion is used for similar reasons to the previous one, and to be able to observe trends in the evolution of their quality. Moreover, this number of revisions provides an adequate set of repeated measures as input to the statistical analysis.

4.3 Data Collection

To build the dataset for our analysis, we relied on the process described in Section 3. In particular, for each project, we have been able to build a dataset containing: (a) the commit SHA; (b) the committer; (c) the experience of the committer at the current timestamp; and (d) the number of introduced Technical Debt issues by the new code of this commit. Next, starting from the first commit timestamp, we created a 6-month time-window that slides monthly, along the evolution of the project. Based on these time-windows, we have created our units of analysis, as shown in Fig. 2. For example, by considering a project that spans across 22 months (M1-M22), we are able to create 16 units of analysis.

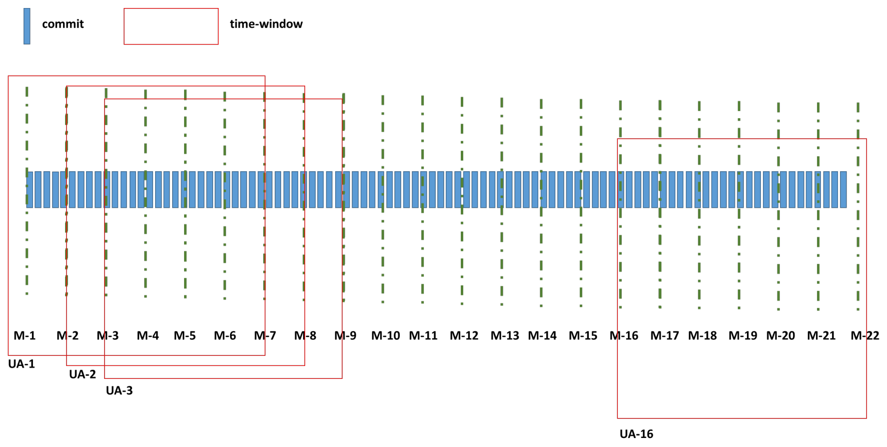


Fig. 2: Demarcating Units of Analysis (sliding temporal windows)

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

With regards to the developers experience it should be highlighted that experience is not a directly observable construct [28]. As a result, experience can be operationalized through multiple variables and be measured in different ways. For example, experience has been measured in terms of years of programming [8], volume of commits [17,1], years of experience expressed in Likert-scale codes [28], etc.

In this study we need to capture the collective experience of an entire team combining both years of experience and volume of commits. The development team experience is a weighted average of developers' experience, over the contributions of a committer in the specific period, as explained in equations (2) - (4). In particular, first we calculate for each developer his/her experience in days at the beginning of each sliding window. This is obtained by measuring the time from their git registration according to eq. 2. Next, for each time window, we calculate the contribution rate of each developer as the number of commits that a developer has contributed to the project over the total commits in that time window as shown in eq. 3. The weighted development experience for each developer is obtained by multiplying the contribution rate with the experience in days. Finally, the entire team's experience is the sum of each developer's experience (eq. 4). To enable the easy replication of this study, a replication package is available online[3].

$$Experience(days)_{dev_i} = CommitTimestamp - GitRegistration_{dev_i} \tag{2}$$

$$ContributionRate_{dev_i} = \frac{numberOfCommits_{dev_i}}{totalCommits} \tag{3}$$

$$TeamExperience = \sum_{i=1}^{developers} ContributionRate_{dev_i} * Experience_{dev_i} \tag{4}$$

4.4 Data Analysis

Data analysis was performed on the aforementioned raw dataset. To answer RQ_1, for each project, we first assess fluctuation by calculating SMF and basic descriptive statistics of the dependent variable [V_2]. Next, to visualize extreme projects (the most stable and most sensitive), we use a line chart representing the evolution of Technical Debt issues introduced by new code. To identify spikes, along evolution we have used a standard method for identifying extreme outliers in SPSS, as defined in equations (5) or (6). As a final step, we explore, if these spikes are concentrated in the beginning, middle, or end of the project.

$$value > Quartile3 + 3 * InterquartileRange \tag{5}$$

$$value < Quartile1 - 3 * InterquartileRange \tag{6}$$

[3] https://zenodo.org/record/5082041

To answer RQ_2, we performed Spearman correlation analyses between variables $[V_2]$ and $[V_3]$, and between $[V_2]$ and $[V_4]$. The choice of a non-parametric

Table 3: SMF of TD issues on New Code

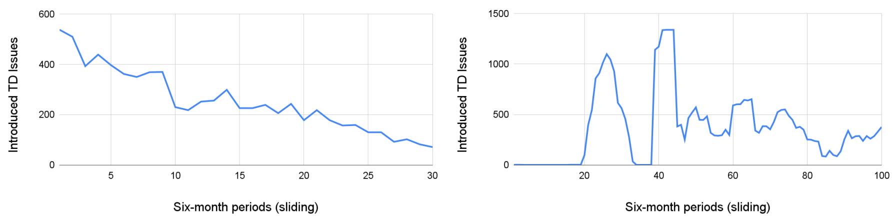
	Project	Mean	Min	Max	Std. Dev.	SMF
Apache Software Foundation	Atlas	475.551	91	1548	407.787	0.538
	Beam	786.081	174	1499	300.519	0.509
	Calcite	470.737	7	2736	692.867	11.902
	Cayenne	94.931	9	353	89.289	1.019
	Commons IO	7.417	0	51	11.638	1.344
	CXF	270.382	0	1008	250.760	0.762
	DeltaSpike	23.936	0	60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

test, was based on the fact that for some projects, the pre-conditions of parametric tests were not met. For extreme sensitive cases we visualize the relation through scatter-plots, and present the co-evolution of the number of commits and the number of Technical Debt issues in a single line chart. For extreme stable cases, we visualize the relation between the average development team experience and the number of Technical Debt issues in a single line chart.

5 Results

5.1 Fluctuation of TD introduction along evolution (RQ1)

In Table 3, we present the fluctuation analysis for the number of TD issues introduced by new code. Based on this, for 28 out of 47 projects the number of TD issues introduced by new code can be considered as stable (dark cells), whereas for the rest 19 as sensitive (light grey cells). For ASF, the percentage of stable projects is 55%, for EF 38%, and for isolated projects 83%; whereas the mean values do not differ in a statistically significant manner (ANOVA F: 1.811 and sig: 0.17). By comparing the SMF of the two ecosystems, we observe that the percentage of stable and sensitive projects is similar, whereas the one for isolated projects is substantially larger. A tentative explanation of this is the existence of specific development guidelines in large OSS ecosystems (like Apache or Eclipse), which limit the *"Bazaar"* effect of collaborative development in "*random*" open-source projects.



(a) `Metron` - "*Stable*" (b) `SIS` - "*Sensitive*"

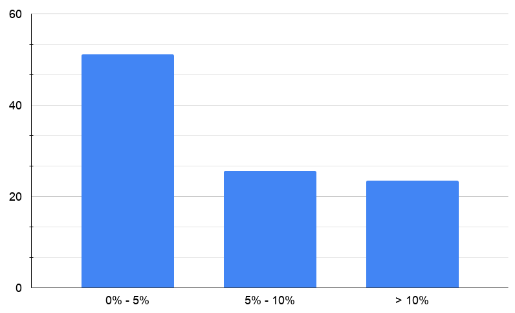
Fig. 3: Indicative project evolution

To provide a visual insight on the discussed fluctuations, in Fig. 3, we present the evolution of one stable project, namely `Metron`, and a sensitive one, namely `SIS`. We note that even for the most "stable" projects, some spikes still exist; however, the spikes are small in height. To study the spikes in more depth, we first calculate their frequency – see Figure 4a. Based on the bar chart, for 51% of the projects the spikes appear in less than 5% of the commits; whereas 23% of the projects produce fluctuations in more than 10% of their commits. We note that the classification of the projects in the three groups (i.e., [0% - 5%],(5% - 10%], and more than 10%) was data-driven. The spikes percentage

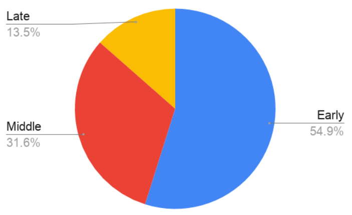
Table 4: Correlation of Commit Activity and TD Introduction

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

for the analyzed projects was less than 15% for 46 out 47 projects in our dataset. Based on this, we split equally the margin of values to three equal ranges [0% - 5%], (5% - 10%], and (10% - 15%]. Since the number of projects with spikes more than 15% was quite low, we have merged those to the last class.
Additionally, in Figure 4b, we observe that fluctuations of TD are distributed across the entire project lifetime (as early, we consider the first third of the studied time windows, as middle the second third, and as late the most recent third). Fluctuations are reduced along evolution, and this decrease is statistically significant, based on the results of performing Friedmans' ANOVA (F: 4.91 and sig: 0.008). The aforementioned observations are first indications that these spikes might be relevant to the time period that they appeared, confirming the relation between Technical Debt issues introduction and project maturity. Nevertheless, this finding needs further investigation.



(a) Frequency of Spikes in Commits

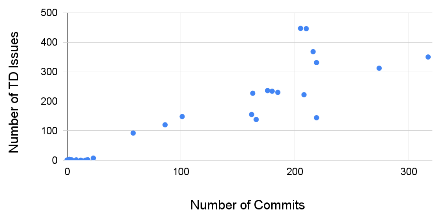


(b) Distribution of Spikes along Evolution

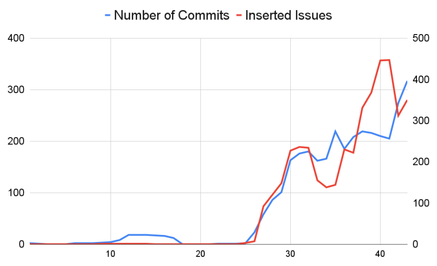
Fig. 4: Project Fluctuations

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

Developers' Activity: One of the first tentative interpretations on the existence of high spikes as those presented in Fig. 3, would be that in the corresponding time windows, lots of code has been committed. To explore the existence of this confounding factor, in Table 4 we highlight with light-gray cell shading (in column Corr. Coef.) the cases in which the correlation is strong (>0.7 [14]) and at the same time statistically significant (p<0.05).The findings suggest that only in 33% of the projects this correlation is strong for ASF, 42% for EF, and 66% for isolated projects. Based on the above, overall, only in 42% of the projects, the commit activity could explain the fluctuations in the number of TD issues that is added by new code. To visualize this result, we present the scatter plot and the evolution of both variables in a single line chart, in Figs. 5a-5b for `Dubbo` (i.e. a project with a high correlation), and in Figs. 6a-6b for `PDFBox` (i.e. a project with a low correlation). In the scatter plots, each dot represents a 6-month period, mapping the values of the two variables for which we seek correlation. For strong correlations, dots would be near to the central diagonal suggesting that a high number of TD issues is observed when the number of commits in a period is also high.

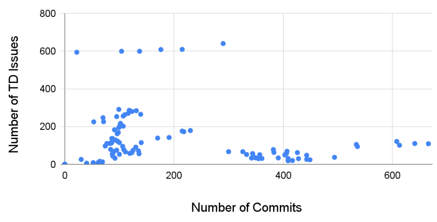


(a) Scatter plot

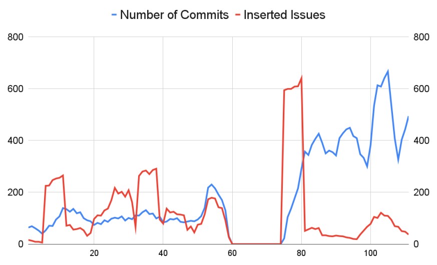


(b) Line chart of co-evolution

Fig. 5: Activity vs. TD Introduction – `Dubbo` project



(a) Scatter plot



(b) Line chart of co-evolution

Fig. 6: Activity vs. TD Introduction – `PDFBox` project

Developers' Experience: Next, we replicate the same correlation process, focusing this time on developers' experience. Similarly to Table 4, in Table 5, we present the correlation coefficient, along with the significance of the correlation between developers' experience and the amount of TD issues introduced in new code. The results suggest that this relation is strong in 14% of the cases for ASF and 0% of the cases for EF, and 16% for isolated projects. This result implies that developers' experience can be 'blamed' only for a limited number of cases of heavy TD introduction. Fig. 7 visualizes cases of strong positive and negative correlations. Finally, we note the sign of the correlation is negative in 66% of the cases, suggesting that experienced teams are producing less TD issues. However, 50% of strong correlations are positive, implying that the cases in which experienced teams are introducing heavy technical debt are not negligible.

Table 5: Correlation of Developers' Experience and TD Introduction

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

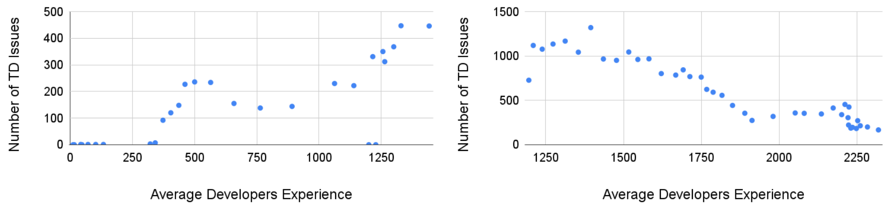


Fig. 7: Experience vs. TD Introduction–Left: `Dubbo`, Right: `TinkerPop`

6 Discussion

6.1 Interpretation of Results

The high-level goal of this study was to investigate if the introduction of Technical Debt issues (by adding new code) is a temporal phenomenon, that diverges over time. Based on the findings, some temporality can be claimed only for a number of projects. In particular, based on the fluctuation of Technical Debt issues due to the introduction of new code (see Section 5.1), we can classify the projects in three categories through visual inspection of the evolution graphs: (a) stable projects without any temporality—i.e., negligible fluctuations (spikes in less than 5% of commits); (b) stable projects that are not sensitive, but some "extra-ordinary" spikes occur (spikes in 5% - 10% of commits); and (c) sensitive projects (spikes in more than 10% of commits).

Based on the findings of RQ1, we can claim that the introduction of Technical Debt issues due to the insertion of new code is independent of time, for more than half of the projects (36 out of 47). This can be interpreted as an indication of project maturity, in the sense that consistent quality is achieved throughout evolution. However, even for these stable projects, the absence of fluctuations does not necessarily imply the absence of any trend. For example, in Fig. 3 we can see that the evolution of project `Metron` does not exhibit any spikes; however, its trend is clearly a decreasing one.

On the other hand, for a subset of the analyzed projects (12 out of 47), the introduction of new code Technical Debt issues is a temporal phenomenon, since many spikes exist in their evolution. For these projects, the number of introduced Technical Debt issues in each period is not stable, and it is reasonable to assume that it is influenced by some external parameters. This can be interpreted as a consequence of the dynamic nature of software development environments, where many factors change along time. This finding is contradicting previous research on the evolution of smells (summarized in Table 1) implying that smells tend to increase monotonically over time. Consequently, it is important to study potential confounding factors that drive the accumulation of Technical Debt issues along the evolution of a software project. A starting point for such factors can be the causes of TD outlined in the studies of Table 1). To some extent, this has been addressed in RQ2.

The second research question lead to rather unexpected findings: i.e., the number of commits made in a time period as well as the experience of developers, are not correlated to the number of introduced Technical Debt issues into the system (for the majority of the cases). Intuitively, one would expect that these variables would be related, in the sense that: (a) the more code is added, the more Technical Debt issues are expected to be introduced; and (b) inexperienced developers would introduce more TD. However, these two factors do not appear to play a key role in the introduction of technical debt. Instead, other confounding factors may be more relevant, such as: (a) the maturity of the project; (b) the developers' habits; or (c) the specific type of tasks performed in each time period. These observations are more evident for the projects that come from the ASF or EF ecosystems, rather than isolated projects. We discuss these directions for further investigation in Section 6.2.

6.2 Implications to Researchers and Practitioners

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

– For stable projects, researchers can further investigate the relation between the stable rate of introduction of new code Technical Debt issues with the practices followed by the developers. It would also be valuable to compare stable projects, but with different trends (increasing vs. decreasing), with respect to their key properties.
– For sensitive projects, researchers can perform explanatory studies to unveil the reasons for which spikes occur in the evolution of the introduced technical debt. Such studies could identify possible reasons (e.g., changes in used libraries or frameworks, impact of business goals) that lead teams/projects with a rather stable accumulation of technical debt, to perform worse under certain circumstances. Furthermore, TD fluctuation can be studied in relation to the number of bugs and issues of an evolving software project, investigating whether spikes in TD introduction cause an excessive number of defects. Based on the findings of our study, such questions would be more easily answered in ASF or EF projects, since for their case the correlation of fluctuation and commit density is lower, leaving more space for exploring other parameters.

- Based on the output of the above, researchers can work on more accurate technical debt prevention methodologies that will attack the heart of the problem, based on the particular conditions of each project. For example, a project that is expected to undergo staff turnover, or will face tight deadlines, should calibrate its quality gates to ensure technical debt does not grow beyond thresholds.

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

7 Threats to Validity

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

Construct Validity reflects how far the examined phenomenon is connected to the intended objectives. The main threat is related to the accuracy by which technical debt can be captured by static analysis tools such as SonarQube. Rule violations reported as Technical Debt issues are only one manifestation of actual code and design inefficiencies. Furthermore, it is known that such tools are not capable of identifying architectural problems or other types of technical debt such as requirements, test or build debt. In addition, we consider only technical debt that can be mapped to methods, thus ignoring changes which might occur at the level of files. However, while SonarQube is by far not perfect in identifying technical debt, other static analysis tools suffer from similar limitations. Another construct validity threat is related to the use of the number of commits as a surrogate of the workload that has been carried out by the project participants. Since in open-source projects, voluntary contribution is interleaved with the rest of the developers' activities, we acknowledge that a 'busy' or 'relaxed' period in terms of commits, does not necessarily reflect the actual work conditions of the developers. Moreover, commits differ in the amount of work that they carry: some commits might be accompanied by many changes in several files while other are related to only a few changes. A final threat to construct validity stems from the current calculation of development experience. In practice, in this work we consider as the starting date of someone's programming career, his/hers registration to GitHub. Although this might be accurate in some cases, we downgrade the experience of developers who have started developing before contributing to GitHub–i.e., started developing in commercial products, or contributed to other OSS repositories. In addition, this calculation is threatened by the fact that the time between registration and commit is not experience equivalent, in the sense that a developer might be idle for a while within the period of interest. Therefore, further research is required to derive the actual workload of developers committing to an open-source software project; as well as their development experience in the start of the sliding window.

Reliability reflects whether the study has been conducted and reported in a way that others can replicate it and reach the same results. To mitigate this threat, the study protocol is explicitly described listing all data collection and analysis steps. The only subjective data interpretation concerns the identification of spikes (which however is of secondary importance); therefore, to a large extent, researcher bias has been avoided. A replication package[3] is available with all available data to allow for an independent replication of the investigation.

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

8 Conclusions and Future Work

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

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

The current study focused only on the impact of new code on the accumulation of TD. However, TD can be introduced or removed along method modification as well. Modified methods are much more challenging to analyze as they entail multiple types of changes such as code addition, removal, modification and refactoring. Nevertheless, it would be of great interest to study the impact of new vs. modified code on the system technical debt along software evolution. Furthermore, the study of the relation between introduced TD issues and developer experience and workload can be performed at a finer-grain level (i.e., introduction of TD at commit level). Analyzing TD at commit level along with the study of the corresponding commit messages could also shed light into the reasons that lie beneath the excessive introduction of TD issues, with the potential of revealing some of the root causes of TD accumulation. In addition to this, the analysis of TD at commit level will unveil how the values of experience and workload vary across sliding-time windows; this could provide further insights and explain the reasons for detecting no correlation between these factors and TD introduction.

References

1. AlOmar, E.A., Peruma, A., Newman, C.D., Mkaouer, M.W., Ouni, A.: On the relationship between developer experience and refactoring. In: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops. ACM (2020). DOI 10.1145/3387940.3392193. URL `https://doi.org/10.1145/3387940.3392193`
2. Alves, V., Niu, N., Alves, C., Valença, G.: Requirements engineering for software product lines: A systematic literature review. Information and Software Technology **52**(8), 806–820 (2010). DOI 10.1016/j.infsof.2010.03.014
3. Amanatidis, T., Mittas, N., Moschou, A., Chatzigeorgiou, A., Ampatzoglou, A., Angelis, L.: Evaluating the agreement among technical debt measurement tools: building an empirical benchmark of technical debt liabilities. Empir. Softw. Eng. **25**(5), 4161–4204 (2020). DOI 10.1007/s10664-020-09869-w. URL `https://doi.org/10.1007/s10664-020-09869-w`
4. Ampatzoglou, A., Ampatzoglou, A., Chatzigeorgiou, A., Avgeriou, P., Abrahamsson, P., Martini, A., Zdun, U., Systa, K.: The perception of technical debt in the embedded systems domain: an industrial case study. In: 2016 IEEE 8th International Workshop on Managing Technical Debt (MTD), pp. 9–16. IEEE (2016)
5. Arvanitou, E.M., Ampatzoglou, A., Chatzigeorgiou, A., Avgeriou, P.: Software metrics fluctuation: a property for assisting the metric selection process. Information and Software Technology **72**, 110–124 (2016). DOI 10.1016/j.infsof.2015.12.010
6. Avgeriou, P.C., Taibi, D., Ampatzoglou, A., Arcelli Fontana, F., Besker, T., Chatzigeorgiou, A., Lenarduzzi, V., Martini, A., Moschou, N., Pigazzini, I., Saarimaki, N., Sas, D.D., de Toledo, S.S., Tsintzira, A.A.: An overview and comparison of technical debt measurement tools. IEEE Software pp. 0–0 (2020). DOI 10.1109/MS.2020.3024958
7. Chatzigeorgiou, A., Manakos, A.: Investigating the evolution of code smells in object-oriented systems. Innovations in Systems and Software Engineering **10**(1), 3–18 (2014)
8. Dieste, O., Aranda, A.M., Uyaguari, F., Turhan, B., Tosun, A., Fucci, D., Oivo, M., Juristo, N.: Empirical evaluation of the effects of experience on code quality and programmer productivity: an exploratory study. Empirical Software Engineering **22**(5), 2457–2542 (2017). DOI 10.1007/s10664-016-9471-3. URL `https://doi.org/10.1007/s10664-016-9471-3`

9. Digkas, G., Ampatzoglou, A., Chatzigeorgiou, A., Avgeriou, P.: On the temporality of introducing code technical debt. In: International Conference on the Quality of Information and Communications Technology, pp. 68–82. Springer (2020)
10. Digkas, G., Chatzigeorgiou, A.N., Ampatzoglou, A., Avgeriou, P.C.: Can clean new code reduce technical debt density. IEEE Transactions on Software Engineering pp. 1–1 (2020). DOI 10.1109/TSE.2020.3032557
11. Digkas, G., Lungu, M., Chatzigeorgiou, A., Avgeriou, P.: The evolution of technical debt in the apache ecosystem. In: European Conference on Software Architecture, pp. 51–66. Springer (2017)
12. Dueñas, J.C., Cuadrado, F., Santillán, M., Ruiz, J.L., et al.: Apache and eclipse: Comparing open source project incubators. IEEE software **24**(6), 90–98 (2007)
13. Falleri, J.R., Morandat, F., Blanc, X., Martinez, M., Monperrus, M.: Fine-grained and accurate source code differencing. In: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, pp. 313–324 (2014)
14. Field, A.: Discovering Statistics Using IBM SPSS Statistics. Sage Publications Ltd. (2013)
15. Hassan, A.E.: Predicting faults using the complexity of code changes. In: 2009 IEEE 31st International Conference on Software Engineering, pp. 78–88 (2009). DOI 10.1109/ICSE.2009.5070510
16. Kazman, R., Cai, Y., Mo, R., Feng, Q., Xiao, L., Haziyev, S., Fedak, V., Shapochka, A.: A case study in locating the architectural roots of technical debt. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 2, pp. 179–188. IEEE (2015). DOI 10.1109/ICSE.2015.146
17. Krutz, D.E., Munaiah, N., Peruma, A., Mkaouer, M.W.: Who added that permission to my app? an analysis of developer permission changes in open source android apps. In: 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft), pp. 165–169. IEEE (2017)
18. Lehman, M.M.: Laws of software evolution revisited. In: European Workshop on Software Process Technology, pp. 108–124. Springer (1996)
19. Letouzey, J.L.: The sqale method for evaluating technical debt. In: 2012 Third International Workshop on Managing Technical Debt (MTD), pp. 31–36. IEEE (2012). DOI 10.1109/MTD.2012.6225997
20. Li, Z., Avgeriou, P., Liang, P.: A systematic mapping study on technical debt and its management. Journal of Systems and Software **101**, 193–220 (2015)
21. Martini, A., Bosch, J., Chaudron, M.: Investigating architectural technical debt accumulation and refactoring over time: A multiple-case study. Information and Software Technology **67**, 237–253 (2015). DOI 10.1016/j.infsof.2015.07.005
22. Olbrich, S., Cruzes, D.S., Basili, V., Zazworka, N.: The evolution and impact of code smells: A case study of two open source systems. In: 2009 3rd international symposium on empirical software engineering and measurement, pp. 390–400. IEEE (2009)
23. Pérez, B., Brito, J.P., Astudillo, H., Correal, D., Rios, N., Spínola, R.O., Mendonça, M., Seaman, C.: Familiarity, causes and reactions of software practitioners to the presence of technical debt: a replicated study in the chilean software industry. In: 2019 38th International Conference of the Chilean Computer Science Society (SCCC), pp. 1–7. IEEE (2019)
24. Peters, R., Zaidman, A.: Evaluating the lifespan of code smells using software repository mining. In: 2012 16th European Conference on Software Maintenance and Reengineering, pp. 411–416. IEEE (2012)
25. Ramač, R., Mandić, V., Taušan, N., Rios, N., de Mendonca Neto, M.G., Seaman, C., Spínola, R.O.: Common causes and effects of technical debt in serbian it: Insightd survey replication. In: 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 354–361. IEEE (2020)
26. Rios, N., Spínola, R.O., Mendonça, M., Seaman, C.: The most common causes and effects of technical debt: first results from a global family of industrial surveys. In: Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 1–10 (2018)
27. Runeson, P., Host, M., Rainer, A., Regnell, B.: Case study research in software engineering: Guidelines and examples. John Wiley & Sons (2012)

28. Siegmund, J., Kästner, C., Liebig, J., Apel, S., Hanenberg, S.: Measuring and modeling programming experience. Empirical Software Engineering **19**(5), 1299–1334 (2013). DOI 10.1007/s10664-013-9286-4. URL `https://doi.org/10.1007/s10664-013-9286-4`
29. Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., Poshyvanyk, D.: When and why your code starts to smell bad (and whether the smells go away). IEEE Transactions on Software Engineering **43**(11), 1063–1088 (2017)