On the Temporality of Introducing Code Technical Debt

Authors field intentionally left blank

Affiliations intentionally left blank

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 work-load, tight delivery schedule, unawareness of good practices, etc. To shed light into the context that leads to technical debt accumulation, in this paper we investigate: (a) the temporality of code technical debt introduction in new methods, i.e., whether the introduction of technical debt is stable across the lifespan of the project, or if its evolution presents spikes; and (b) the relation of technical debt introduction and the development team's workload in a given period. To answer these questions, we perform a case study on twenty-seven Apache projects, and inspect the number of Technical Debt Items introduced in 6-month sliding temporal windows. The results of the study suggest that: (a) overall, the number of Technical Debt Items introduced through new code is a stable metric, although it presents some spikes; and (b) the number of commits performed is not strongly correlated to the number of introduced Technical Debt Items.

Keywords: technical debt 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 [1]. These inefficiencies manifest themselves as violations of coding standards, complex and hard to understand code, code duplicates, etc. [2]. According to Alves et al. [3] code TD is the most studied type of technical debt, and based on Ampatzoglou et al. [4] it is one of the most important in industry.

There has been significant work on how code TD evolves and how it accumulates over time. However, existing studies have looked at TD evolution as a whole, without distinguishing between Technical Debt that is added as new code, and TD that is added or modified in existing code. In this paper, we focus only on the introduction of new code TD, i.e. TD inserted in the system in the form of new Technical Debt Items (TDIs). More specifically we study new methods (our scope is object-oriented systems) that contain TD and we look at the introduction of this type of new TD as a temporal phenomenon.

Focusing on TD that is introduced by new code, as opposed to TD that is introduced by modifying existing code, can provide a unique insight. Specifically,

the new TDIs 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 TD through a clearer source.

In particular, we explore: (***1***) if the number of introduced TDIs is uniformly spread across evolution, or whether there are time windows in which more TDIs are inserted; and (***2***) if the number of TDIs that is introduced along evolution is related to the activity (intensity of commits) of developers in different time windows. Projects could exhibit either a stability in the introduction of code TDIs across evolution or experience fluctuations with isolated or repeating spikes of introduced code TDIs. In the former case one could assume that accumulation of TD is most probably due to factors that are constantly present in the entire lifetime of the project, such as employees' skills, used methodologies, tools, management practices, etc. In the latter case, one could postulate that the insertion of new code TDIs is a highly temporal phenomenon depending on volatile factors such as feature requests, changing schedules, pressure to fix bugs, etc.

To achieve this goal, we explore the evolution of twenty-seven projects by the Apache Software Foundation, and we track the number of new TDIs inserted in each commit. Next, we create a 6-month sliding window, and we calculate the cumulative number of inserted TDIs for each window, as well as the number of commits in the same time period. To answer the first question, we use a metric property (termed SMF—see Section 3.2) that is able to assess metrics fluctuation along time and characterize them as either stable or sensitive. To answer the second question we correlate the number of commits for each window to the number of inserted TDIs. The reporting and interpretation of the results is performed at the project level.

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

Many studies have explored the evolution of code quality, and the reasons for its degradation. Since this paper focuses on the introduction of TD over time, we organize this sub-section into causes of TD introduction and TD evolution.

Causes of Technical Debt Introduction: Tufano et al. [5] 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. [6] who conducted a case study on the roots of architecture debt, Architectural Technical Debt (ATD) is extremely common and probably the most important type of TD 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.

Martini et al. [7] 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.

Evolution of Technical Debt: Although TD is a multifaceted concept, one of the key constituents of code TD is the presence of code smells. One of the first studies that investigate the evolution of code smells was conducted by Olbrich et al. [8]. 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 [9] 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 [10] 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 TD in sixty-six open-source Java projects by the ASF, over a period of 5 years. In order to detect issues that incur TD, they relied on SonarQube. The results show that on the one hand, there is a significant increasing trend on the size, complexity, number of TDIs, and the total TD over time, which seems to confirm the software aging phenomenon. But on the other hand, when TD 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 TDIs; or (b) developers that introduce better quality code in each commit (compared to the project's existing code base).

Despite the fact that code TD introduction has been widely explored, we lack evidence on: (a) the way in which TD is introduced, i.e. whether there is stable increase, or large fluctuations exist, and (b) if such fluctuations coincide with large-scale changes in the codebase.

3 Background Information

In this section we present information that is necessary for understanding the paper. In Section 3.1 we provide an overview of the process for identifying new TDIs in each commit. In Section 3.2 we present an overview of the approach for assessing metrics fluctuation; this is employed to assess the stability in the introduction of new code TDIs over time.

3.1 Identifying New TD Items along Evolution

To analyze software systems and measure TD throughout their evolution, we have used SonarQube 7.9.2 LTS. SonarQube (SQ) relies on a set of rules which are checked by static source code analysis; every time a piece of code breaks one of those coding or design rules, a Technical Debt Issue is raised. SQ estimates the effort (in minutes) required to eliminate the identified TDIs. This effort is obtained by assigning a time estimate for fixing each type of problem and by multiplying the number of all TDIs of that type with that estimate.

Considering that software systems evolve through a number of revisions and that in each revision several types of changes may occur simultaneously, we look at the three major types of code changes: the introduction of new code, the deletion and the modification of existing code. In this paper we work at the method level, that is, we aggregate all TDIs reported by SQ for individual lines to the method in which they belong. The reason for this decision is that monitoring changes at the instruction level would be more complex and less accurate considering that several types of changes can simultaneously occur in some statements (e.g., modification and introduction of new code). Furthermore, tracking changes at the instruction level is challenging, as one would have to map each instruction (in a particular revision) to the corresponding instruction in the previous revision. This process is complicated by the insertion of new statements, comments, blank lines, etc. Therefore, to be certain about the classification of changes, we monitor changes at the method level.

At each revision a class can be added, deleted, modified, renamed or remain unchanged. The same applies for the methods. As explained above, we only focus on the introduced TDIs in the newly inserted methods. A new method can be added either in an existing class or upon the creation of a new class. To distinguish the newly inserted methods for each commit from the deleted, modified, renamed, and unchanged ones, we rely on the Gumtree Spoon AST Diff tool [12]. For each revision, first, we detect all changes that occurred in the corresponding commit at the file-level, i.e. we identify the added, modified,

renamed, and deleted files. Then, we exclude the deleted files which do not exist anymore in the examined commit. For the added files/classes, we consider all methods as new code; in other words we consider them as newly inserted methods in new classes. For the modified and renamed files we compare their AST with the AST in the previous revision (using the Gumtree Spoon tool). By this comparison we identify the newly inserted methods in existing classes.

After identifying which methods have been inserted into the project (in the commit under study) and their span (starting/ending line in the file), we can further identify TDIs. For this step we analyze the project using SQ. Then, we retrieve all the TDIs (via SonarQube's API) and keep only the ones that can be mapped to the newly inserted methods. This is performed by matching the line in which each TDI is reported by SQ with the method containing that line.

3.2 Fluctuation of Software Metrics

Software Metrics Fluctuation (SMF) is a property of metrics, defined as "*the degree to which a metric score changes from one version of the system to the other*" [13]. Using SMF, metrics can be characterized as ***sensitive*** (changes induce high variation on the metric score) or ***stable*** (changes induce low variation). To capture the SMF property of a metric, that property should:

- Take into account the order of measurements in a metric time series. This is the main characteristic that a fluctuation property should hold, in the sense that it should quantify the extent to which a score changes between two subsequent time points.
- Yield values that can be intuitively interpreted, especially for border cases. Therefore, if a score does not change at all, its fluctuation should be evaluated to zero. Any other change pattern should result in a non-zero fluctuation value. Finally, the highest value should be obtained for time series that constantly change and fluctuate between the two ends of their range, for every pair of successive versions of the software.

To assess SMF, in this paper, we use a measure proposed by Arvanitou et al. [13], namely `mf`. The measure is defined as: "*the average deviation from zero of the difference ratios between every pair of successive versions*", as shown below.

$$mf = \sqrt{\frac{\sum_{i=2}^{n} \left(\frac{score_i - score_{i-1}}{score_{i-1}}\right)^2}{n-1}}$$

In the study that introduced SMF [13], the authors also explored various alternatives (such as coefficient of variance, and auto-correlation-of-lag-one), which however, were not able to capture the aforementioned properties of SMF.

4 Case Study Design

In this section, we present the design of the case study which was based on the linear-analytic structure as described by Runeson et al. [14].

4.1 Research Questions

As already mentioned in the Introduction Section, we ask two research questions.

RQ_1: *Does the number of introduced technical debt items by new code fluctuate along evolution?*

The answer to this research question will unveil if in different time periods, different amounts of TD are introduced. The answer reflects the main goal of this study, i.e., to investigate the temporality of the TD phenomenon. Specifically, this answer will enable us to characterize TDIs introduction as either stable, or sensitive to temporal influence. In addition, we will study any possible spikes in the evolution on new code TD, 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 introduced technical debt items by new code, correlate to the activity of developers?*

To increase the confidence in the results of the previous research question, we study a potentially important confounding factor for this empirical setup: developers' activity. 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 TDIs could be due to more intense programming activity in the corresponding periods.

4.2 Cases and Units of Analysis

This study is characterized as a multiple, embedded case study [14], 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 amount of code TDIs 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 OSS systems is the vast amount of data that is available in OSS repositories, 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 only high-quality projects that evolve over a period of time, we looked into Apache projects and investigated the projects presented in Table 1. 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.
- The software contains more than 100 classes to ensure the inclusion of systems with a substantial size, functionality and complexity.

Table 1: Selected Projects

Project	Classes	NCLOC	Analyzed Revisions	Project	Classes	NCLOC	Analyzed Revisions
Atlas	932	87637	1454	Knox	1083	51429	1033
Beam	3757	176663	2780	Kylin	1658	128531	3205
Calcite	2606	186633	1448	Metron	1433	72579	548
Cayenne	2615	164170	2116	MyFaces	1843	174158	1211
Commons IO	132	10500	1059	NiFi	4256	371031	1490
CXF	4111	353085	5079	oozie	1082	97597	587
DeltaSpike	951	46182	513	OpenWebBeans	561	44299	1583
Drill	4655	316552	1316	PDFBox	1279	136916	3758
Dubbo	943	61865	728	Pulsar	1837	147182	1503
Flink	5632	341149	5329	SIS	1948	181588	828
Flume	790	51897	789	Storm	3958	243574	738
Giraph	1414	72972	668	TinkerPop	1698	95652	5178
Jackrabbit	2883	273574	4260	Zeppelin	1209	89193	1562
jclouds	5687	227459	4323				

– The software has more than 1000 commits. We have included this criterion for similar reasons to the previous criterion 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 3.1. In particular, for each project, we have been able to build a dataset containing: (a) the commit SHA; (b) the commit timestamp; and (c) the number of introduced TDIs 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. 1. For example, by considering a project that spans across 22 months (M1-M22), we are able to create 16 units of analysis.

For each period captured by the time-window, we summed the number of TDIs that were introduced in all commits included in the timeframe. Therefore, the final dataset consists of three variables: [V_1] time-window (in months/year); [V_2] number of commits in the time-window; and [V_3] number of TDIs introduced by new code in the time-window. A replication package is available online[1].

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

[1] https://drive.google.com/drive/folders/1oF51ZPlXSiIL-mM-W2kHs7vi63Ij5n8P

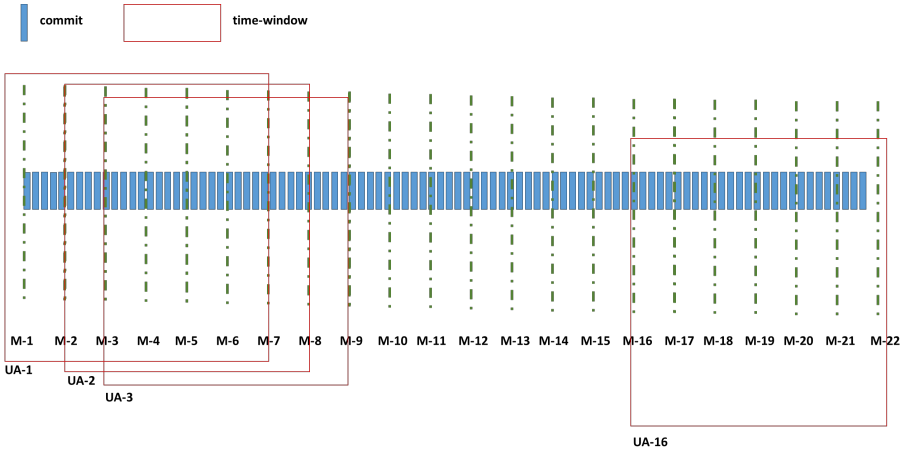


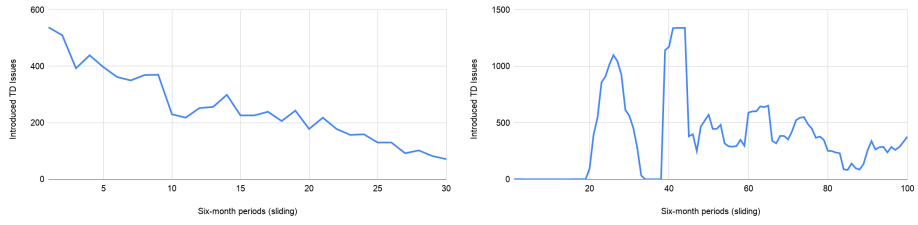
Fig. 1: Demarcating Units of Analysis (sliding temporal windows)

descriptive statistics of the dependent variable [V_3]. Next, to visualize extreme projects (the most stable and most sensitive), we use a line chart representing the evolution of TDIs introduced by new code. By inspecting the line chart, we highlight spikes in the introduction of Technical Debt Issues, and discuss, if they seemed more concentrated in the beginning, middle, or end of the project. To answer RQ_2, we performed Pearson correlation analyses, and for extreme cases we visualize the relation through scatterplots, and present the co-evolution of number of commits and the number of TDIs in a single line chart.

5 Results

5.1 Fluctuation Analysis (RQ1)

In Table 2, we observe the results of the fluctuation analysis for the number of TDIs introduced by new code, in the 27 cases of the study, based on the value of the Software Metrics Fluctuation metric. We can observe that for 16 out of 27



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

Fig. 2: Indicative project evolution

Table 2: TD Fluctuation per Project

Project	SMF	Corr. Coef.	Sig. Level	Spk	Project	SMF	Corr. Coef.	Sig. Level	Spk
Atlas	0.538	0.500	0.000	1	Knox	0.301	0.361	0.002	2
Beam	0.509	0.502	0.002	3	Kylin	0.343	0.598	0.000	3
Calcite	11.902	0.150	0.195		Metron	0.162	0.551	0.002	0
Cayenne	1.019	0.584	0.000		MyFaces	2.992	0.355	0.000	
Commons IO	1.344	0.661	0.000		NiFi	0.024	0.302	0.073	1
CXF	0.762	0.363	0.000	3	oozie	0.451	0.198	0.075	1
DeltaSpike	1.396	0.791	0.000		jclouds	0.467	0.890	0.000	1
Drill	0.335	0.519	0.001	1	PDFBox	3.505	−0.066	0.493	
Dubbo	1.900	0.929	0.000		Pulsar	0.456	0.768	0.000	1
Flink	4.080	0.353	0.001		SIS	9.558	0.482	0.000	
Flume	0.340	0.922	0.000	1	Storm	0.389	0.071	0.611	2
Giraph	1.174	0.463	0.000		TinkerPop	0.156	0.802	0.000	1
Jackrabbit	1.639	0.453	0.000		Zeppelin	0.320	0.161	0.220	2
OpenWebBeans	0.492	0.436	0.000	1					

Spk = Spikes

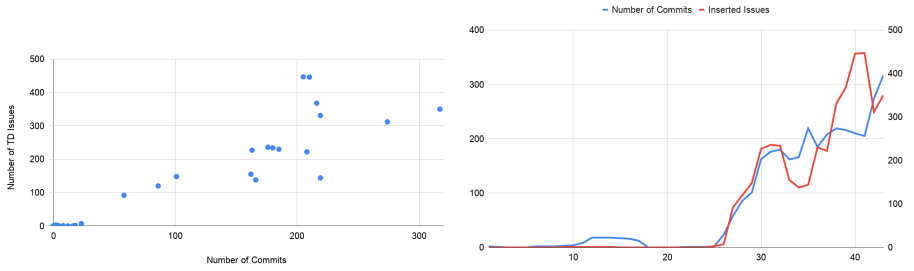
projects the metric under study (i.e., number of TDIs introduced by new code) can be considered as stable (dark grey cell shading in column SMF), whereas in the rest 11 projects as sensitive (light grey cell shading).

To provide a visual insight on the discussed fluctuations, in Fig. 2, we present the evolution of one extremely 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. A visual analysis of fluctuations in all projects (figures are available in the online material) revealed that fluctuations of TD are distributed across the entire project lifetime. This observation is a first indication that these spikes might be irrelevant to the time period that they appeared, questioning a relation between TD introduction and project maturity. Nevertheless, this finding needs further investigation.

5.2 Correlation Analysis: Fluctuation vs. Activity (RQ2)

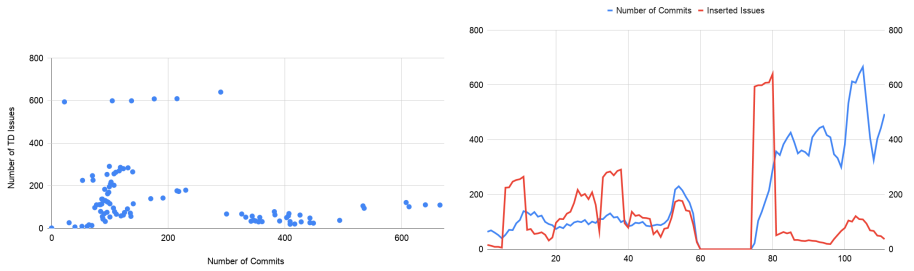
To investigate if the fluctuation of the number of TDIs that is inserted by new code is due to some temporal phenomenon that occurs in the given time period, we need to exclude the most obvious confounding factor, i.e., developers' activity. One of the first tentative interpretations on the existence of high spikes as those presented in Fig. 2(b), would be that in the corresponding time windows, lots of code has been committed. To explore the existence of this confounding factor, in Table 2 we highlight with light-gray cell shading (in column Corr. Coef.) the cases in which the correlation is strong (>0.7 [15]) and at the same time

statistically significant ($p<0.001$). The findings suggest that only in 22% of the projects this correlation is strong. So only in these cases, the commit activity could explain the fluctuations in the number of TDIs 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. 3a-3b for `Dubbo` (the project with the highest correlation), and in Figs. 4a-4b for `PDFBox` (the project with the lowest 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 are expected to concentrate around the central diagonal.



(a) Scatter plot (b) Line chart of co-evolution

Fig. 3: Correlation analysis for `Dubbo`: fluctuation related to developers' activity



(a) Scatter plot (b) Line chart of co-evolution

Fig. 4: Correlation analysis for `PDFBox`: fluctuation NOT related to developers' activity

6 Discussion

6.1 Interpretation of Results

The high-level goal of this study was to investigate if the introduction of TDIs (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 TDIs 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 (0-1 spike, 10 projects); (b) stable projects that are not sensitive, but some "extra-ordinary" spikes occur (>1 spikes, 6 projects); and (c) sensitive projects (many spikes, 11 projects). The number of spikes of each project is reported in Table 2 (column 'Spk'); note that we only provide the number of spikes for the stable projects, since sensitive projects have multiple ones.

Based on the findings of Table 2, we can claim that the introduction of TDIs due to the insertion of new code is, in the majority of the projects, independent of time. This can be interpreted as an indication of project maturity, in the sense that consistent quality is achieved throughout evolution. However, even for these projects, the absence of fluctuations does not necessarily imply the absence of any trend. For example, in Fig.2 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, the introduction of new code TDIs is a temporal phenomenon, since many spikes exist in their evolution. For these projects, the number of introduced TDIs in each period is not stable, and it is reasonable to assume that it is influenced by some external parameters. This observation renders important the study of potential external factors that drive the accumulation of TDIs along the evolution of a software project.

The second research question that we have explored led to a rather unexpected finding: i.e., the number of commits, made in a time period, is (for the majority of the cases) not correlated to the number of introduced TDIs into the system. Intuitively, one would expect that these variables would be related, in the sense that the more code is added, the more TDIs are expected to be introduced. However, this might not be the case for several reasons, i.e., TD might be more strongly related to: (a) the maturity of the project; (b) the developers' habits; or (c) the specific type of tasks performed in each time period. Therefore, this issue needs further investigation, as discussed 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 TDIs reflects (at least to some extent) the characteristics of the development process: by being stable in most cases, the introduction of new code TD 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: TDIs 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 that researchers:

- For stable projects, investigate further the relation between the constant rate of introduction of new code TDIs with the practices followed by the

development team. 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, perform explanatory studies to unveil the reasons for which spikes occur in the evolution of the introduced TD. Such studies could identify possible reasons (e.g., changes in the programming team, changes in used libraries or frameworks, impact of business goals) that lead teams/projects with a rather stable accumulation of TD, to perform worse under certain circumstances.
- Based on the output of the above, researchers should work on more accurate TD 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 TD does not grow beyond thresholds.

Regarding ***practitioners***, we suggest the following implications:

- We encourage them to perform fluctuation analysis and investigate the reasons for the existence of high or frequent peaks in the evolution of introduced TDIs. Understanding the consequences of their way of working in certain periods (which might lead to excessive accumulation of TD) can prove beneficial for process improvement purposes and quality control.
- We advise them to classify their project in the categories mentioned in Section 6.1. If their project is sensitive or if the observed trend is a steadily increasing one, then they need to perform a root cause analysis regarding the parameters that affect the accumulation of new code TD. Some of them may be mitigated, for example moving certain developers to different teams, or reprioritizing the backlog to include more refactoring.

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 involved threat is related to the accuracy by which TD can be captured by static analysis tools such as SonarQube. Rule violations reported as TDIs are obviously 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 TD such as requirements, test or build debt. In addition, we consider only TD 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 TD, other static analysis tools suffer from similar limitations.

Another construct validity threat is related to the use of the number of commits as a surrogate of the workload that has been carried out by the project

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

Reliability reflects whether the study has been conducted and reported in a way that others can replicate it and reach the same results. To mitigate this threat, the study protocol is explicitly described listing all data collection and analysis steps. The only subjective data interpretation concerns the identification of spikes (which however is of secondary importance); therefore, to a large extent, researcher bias has been avoided. A replication package[1] 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 TD tools. We have focused only on Java Apache 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 27 projects of the ASF, 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 TD are needed to strengthen the validity of the derived conclusions.

8 Conclusions

Studying the phenomenon of introducing code TDIs is a research direction that is important for building tools aimed at preventing the accumulation of TD. In this study, we focus on code technical debt, and in particular, we explore the temporality of the TD introduction phenomenon. To this end, we explore if the introduction of TDIs changes in different time periods, and if these changes can be attributes to the developers' activity in the corresponding period. To explore these two questions, we have performed a case study on the complete evolution of twenty-seven projects from the ASF.

The results of the study suggested that for the majority of the projects the evolution on TD introduction is stable, i.e., there are not many (at maximum 2) high fluctuations in TDIs introduction, due to new code. However, a non-negligible part of projects (approx. 40%) present high and frequent fluctuations. This result suggest that TD introduction is only partially a temporal phenomenon, with more TD being introduced in some time periods. The additional exploration of the phenomenon led to the conclusion that the spikes in the evolution of TD introduction are not correlated with spikes in the development activity, suggesting that the number of commits in the examined period is not the main factor affecting the existence of 'excessive TD introduction.

References

1. Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *Journal of Systems and Software*, vol. 101, pp. 193–220, 2015.
2. J.-L. Letouzey, "The sqale method for evaluating technical debt," in *2012 Third International Workshop on Managing Technical Debt (MTD)*. IEEE, 2012, pp. 31–36.
3. V. Alves, N. Niu, C. Alves, and G. Valença, "Requirements engineering for software product lines: A systematic literature review," *Information and Software Technology*, vol. 52, no. 8, pp. 806–820, 2010.
4. A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, P. Avgeriou, P. Abrahamsson, A. Martini, U. Zdun, and K. Systa, "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)*. IEEE, 2016, pp. 9–16.
5. M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1063–1088, 2017.
6. R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyev, V. Fedak, and A. Shapochka, "A case study in locating the architectural roots of technical debt," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 179–188.
7. A. Martini, J. Bosch, and M. Chaudron, "Investigating architectural technical debt accumulation and refactoring over time: A multiple-case study," *Information and Software Technology*, vol. 67, pp. 237–253, 2015.
8. S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "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*. IEEE, 2009, pp. 390–400.
9. A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of code smells in object-oriented systems," *Innovations in Systems and Software Engineering*, vol. 10, no. 1, pp. 3–18, 2014.
10. R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, 2012, pp. 411–416.
11. G. Digkas, M. Lungu, A. Chatzigeorgiou, and P. Avgeriou, "The evolution of technical debt in the apache ecosystem," in *European Conference on Software Architecture*. Springer, 2017, pp. 51–66.
12. J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 313–324.
13. E.-M. Arvanitou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou, "Software metrics fluctuation: a property for assisting the metric selection process," *Information and Software Technology*, vol. 72, pp. 110–124, 2016.
14. P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case Study Research in Software Engineering: Guidelines and Examples*. John Wiley & Sons, 2012.
15. A. Field, *Discovering Statistics Using IBM SPSS Statistics*. Sage Publications Ltd., 2013.