The Evolution of Technical Debt in the Apache Ecosystem

Georgios Digkas[1(✉)], Mircea Lungu[1], Alexander Chatzigeorgiou[2], and Paris Avgeriou[1]

[1] Johann Bernoulli Institute for Mathematics and Computer Science, University of Groningen, Nijenborgh 9, 9747 AG Groningen, The Netherlands
{g.digkas,m.f.lungu}@rug.nl, paris@cs.rug.nl
[2] Department of Applied Informatics, University of Macedonia, Egnatia 156, 546 36 Thessaloniki, Greece
achat@uom.gr

Abstract. Software systems must evolve over time or become increasingly irrelevant says one of Lehman's laws of software evolution. Many studies have been presented in the literature that investigate the evolution of software systems but few have focused on the evolution of technical debt. In this paper we study sixty-six Java open-source software projects from the Apache ecosystem focusing on the evolution of technical debt. We analyze the evolution of these systems over the past five years at the temporal granularity level of weekly snapshots. We calculate the trends of the technical debt time series but we also investigate the lower-level constituent components of this technical debt. We aggregate some of the information to the ecosystem level.

Our findings show that the technical debt together with source code metrics increase for the majority of the systems. However, technical debt normalized to the size of the system actually decreases over time in the majority of the systems under investigation. Furthermore, we discover that some of the most frequent and time-consuming types of technical debt are related to improper exception handling and code duplication.

Keywords: Software evolution · Time series data mining · Technical debt · Mining software repositories · Empirical study

1 Introduction

The Technical Debt (TD) metaphor was coined by Ward Cunningham in 1992 as:

"*Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. Objects make the cost of this transaction tolerable. The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise*".

A. Lopes and R. de Lemos (Eds.): ECSA 2017, LNCS 10475, pp. 51–66, 2017.
DOI: 10.1007/978-3-319-65831-5_4

Technical debt is present in both industrial software as well as in open-source projects. In industrial settings the tight deadlines are pushing the software engineers and the developers to compromise the quality of the system and take shortcuts in order to release a product as soon as possible. In open-source settings, the self-imposed deadlines of the developers working towards delivering their contributions to the community or the lack of processes regarding quality assurance might lead them to take similar shortcuts.

Taking all these shortcuts increases the change- and fault- proneness of the systems and aggravates long-term understandability, re-usability, reliability, efficiency, security, and maintainability. While it has been empirically proven that these shortcuts affect negatively the project's quality, completely eliminating technical debt from a system is undesirable as the investment to reduce TD would be extremely inefficient.

Although there has been extensive research with respect to technical debt [1] and there exists even a dedicated international forum for research on the topic (the MTD workshop) there is a lack of empirical evidence regarding the occurrence and evolution of technical debt in the open-source systems.

Moreover, technical debt has also not been studied before in software ecosystems. Software ecosystems are groups of software projects that are developed and co-evolve in the same environment [2,3]. Such projects can share code, they might depend on one another, and are often built on similar technologies and with similar processes.

In this paper we conduct an empirical study of sixty-six open-source software projects of the Apache ecosystem. We chose to analyze OSS projects of the Apache Software Foundation because it is one of the biggest communities which provide software products for the public good and its projects are highly appreciated and used.

Structure of the paper. The rest of the paper is organized as follows: Sect. 2 motivates our study through an analysis of an Apache project. Section 3 presents the methodology and the design of the study. Section 4 presents the results of our empirical study and Sect. 5 discusses the threats to its validity. Section 6 presents the related work and Sect. 7 concludes the paper.

2 A Motivating Example

Apache Sling[1], one of the most popular projects in the Apache ecosystem, is an open-source Web framework for the Java platform designed to create content-centric applications on top of a JSR-170-compliant content repository such as Apache Jackrabbit. The project represents a significant community effort: at the moment of writing this article, the system has more than a dozen contributors who have commit rights to the main repository; these committers have contributed more than twenty thousand commits over the years.

Imagine we are the developers of the Apache Sling. We decide to analyze it to learn about the evolution of its technical debt. To measure the technical debt and

[1] https://sling.apache.org/.

extract other metrics we decide to use an industrial strength tool. SonarQube [4] is an open-source tool for continuous inspection which features dashboards, rule-based defect analysis, and build integration. It supports various languages, including Java, C, C++, C#, PHP, and JavaScript.

SonarQube employs the SQALE [2] method for estimating the time required to fix the technical debt [5]. Technical debt evolution estimation using SonarQube is time consuming since when recomputing it for a new version of the system, no matter how small the difference between the two versions (even a single commit), the tool can not analyze only the differences, but instead, has to do the entire computation for the entire system again[3]. This means that the time necessary to analyze the history of a system is proportional to the number of versions of the system that are to be analyzed.

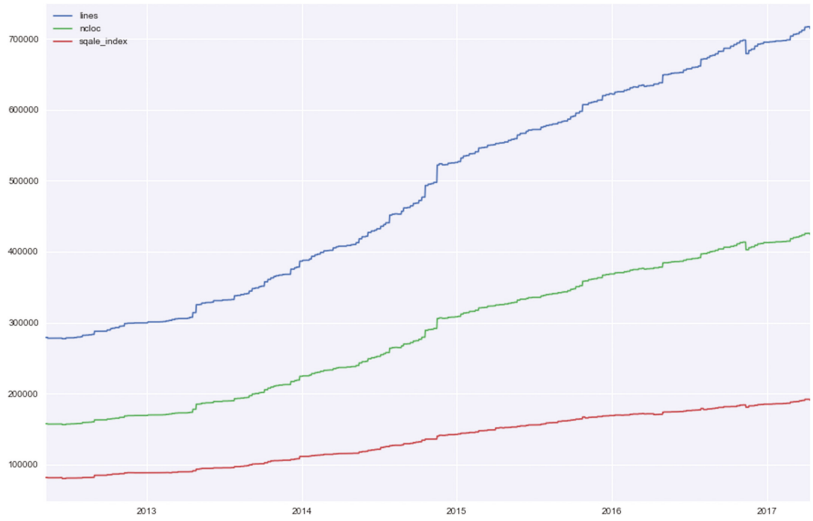

Fig. 1. The evolution of lines of code (blue, top), non commented lines of code (green, middle), and technical debt (red, bottom) in Apache Sling over the last 5 years. Technical debt represents effort to fix problems and is estimated in minutes. (Color figure online)

To drive this evolutionary analysis we do not use the graphical UI of the tool but rather we develop a program that interfaces with the API of the tool in order to compute the entire battery of analyses that SonarQube supports on that version, including technical debt related analysis.

[2] Software Quality Assessment based on Lifecycle Expectations.

[3] This problem is not exclusive to SonarQube. We are not aware of any analysis tools that perform complex, system-level analysis without re-analyzing the entire system when presented with a new version.

However, since analyzing all commits of the Sling project (which are more than twenty thousands) is not feasible as we explained earlier, we analyze snapshots of the system at one week intervals. More precisely we find the last commit in a given week, and we run the analysis on that version.

Figure 1 presents the evolution of the Apache Sling project over the last five years in terms of two metrics:

1. Lines of code (blue, topmost series) and non-comment lines of code (green, middle series)
2. SQALE Index which is the tool estimated technical debt (red, bottom series) in minutes

The most salient observation in the Fig. 1 is that the amount of measured technical debt as measured by the SQALE index grows in parallel to the magnitude of the system as measured in lines of code. This is indeed not surprising as it is well known that as systems age their architecture erodes [6]. Moreover, it is reasonable to assume that the absolute number of identified inefficiencies increases with the amount of functionality delivered by the system.

Fig. 2. The evolution of normalized technical debt in Apache Sling over the last 5 years shows a clear decreasing trend

To exclude the possibility that the growth of technical debt is correlated with the growth of the system, we compute the evolution of the *size-normalized technical debt* – that is, the technical debt normalized to the number of lines of code in the system. Figure 2 shows that the normalized technical debt is actually decreasing.

Open Questions. How this project compares with other similar ones? Is the growth of technical debt that our system exposes normal? Is the fact that the normalized technical debt decreases over time exceptional?

Table 1. The most frequently occurring types of issues in Apache Sling. The number in parenthesis is the percentage of the total violations detected

	Issue	Count	
1	The diamond operator ("<>") should be used	2,888	(10.75%)
2	String literals should not be duplicated	2,875	(10.70%)
3	Generic exceptions should never be thrown	1,856	(6.90%)
4	Control flow statements should not be nested too deeply	1,345	(5.00%)
5	Exception handlers should preserve the original exceptions	1,215	(4.50%)

The Components of Technical Debt. The technical debt estimation is based on what SonarQube calls "rule violations" or *issues*. The tool detects the violations of a large variety of rules for software quality. These issues are classified by the tool into three categories: bugs, violations, and code smells. The version 6.2 of the tool that we used distinguishes among 397 Java rules. It divides them in the following 3 types namely: Bug (150 rules), Vulnerability (31 rules), and Code Smell (216). Table 1 shows the top 5 most frequent issues that are being violated in the case-study system.

Table 2. The most costly to fix types of issues in Apache Sling. The starred issues did not appear in the previous table

	Issue	Time (minutes)	
1	String literals should not be duplicated	39,234	(13.8%)
2	Generic exceptions should never be thrown	37,120	(13.0%)
3	*Source files should not have any duplicated blocks	30,440	(10.7%)
4	*Cognitive complexity of methods should not be too high	16,061	(5.6%)
5	Control flow statements should not be nested too deeply	13,450	(4.7%)

Besides listing issues, SonarQube also estimates the time required to fix them. Table 2 shows the top 5 most time consuming issues as estimated by the tool.

The two tables show that the most frequent violations are not necessarily the most time consuming to fix: issue 1 in Table 2 was ranked lower in Table 1, and issues 3 and 4 did not even appear in that table. It is also interesting to see that some of the detected problems are quite low-level (e.g. string literal duplication) while others are relevant for the higher level architecture of the system (e.g. a seemingly absent exception policy, large scale code duplication).

Open Questions. Are the relative frequency and effort required to fix these issues are characteristic to the Apache Sling project or they are more generally characteristic to Java systems? Is this uneven distribution of effort towards some issues specific or generic?

3 Study Design

Inspired by the open questions presented in the previous section, the *goal* of our study is then, to analyze the evolution of OSS projects in the Apache ecosystem for the *purpose* of understanding and investigating the accumulation of TD and the evolution of source code metrics. More specifically, our study aims at addressing the following four research questions (RQs):

RQ$_1$: *How does the technical debt of the open-source systems in the Apache ecosystem evolve over time?*
The motivation for this question is to investigate the evolution of TD as it is generated by a widely acknowledged tool for a large set of OSS projects belonging to the same ecosystem.

RQ$_2$: *How does the normalized technical debt of the open-source systems in the Apache ecosystem evolve over time?*
Because the amount of TD might be related to the size of the code base this research question aims at investigating the evolution of TD when normalized over the size of each system.

RQ$_3$: *What are the most frequent types of technical debt in the studied ecosystem?*
The motivation for this question is to validate whether developers incur specific types of debt or not

RQ$_4$: *What are the most costly to fix types of technical debt in the studied ecosystem?*
Since the effort required to repay TD varies among violations the goal of this question to obtain an insight into the actual effort to eliminate the most frequent sources of TD.

It is for brevity, that in the research questions and the rest of the paper we talk about *technical debt* but we clearly mean *technical debt as estimated by SQALE method implemented in the SonarQube tool*. The evolutionary study of technical debt as measured and estimated with other tools falls outside of our intended scope for this study.

3.1 Project Selection

The context of the study is the evolution of the Java open-source software projects developed by the Apache Software Foundation. Since the analysis we perform is computationally intensive we limit our study to a sample of sixty-six

Table 3. The list of projects included in the study

Project	NCLOC	Classes	Project	NCLOC	Classes
sling	425,831	6,058	opennlp	62,141	998
zookeeper	74,898	948	chukwa	42,734	577
tomcat60	180,766	1,676	tapestry-5	157,911	3,266
jspwiki	57,967	555	manifoldcf	209,190	1,824
directory-shared	197,377	1,611	crunch	52,564	1,025
cayenne	232,876	3,818	jena	444,414	5,970
commons-collections	61,637	741	oodt	128,875	1,810
openjpa	431,915	5,358	sis	205,367	2,204
mina	23,633	442	commons-csv	5,197	35
poi	367,828	3,907	commons-vfs	33,315	427
nutch	51,738	639	falcon	122,277	1,015
commons-lang	74,849	569	aurora	68,894	1,156
commons-io	29,267	271	jclouds	340,647	6,950
httpclient	61,657	685	helix	81,729	1,060
wicket	211,627	4,175	struts	152,296	2,341
batik	191,790	2,590	cxf	635,020	8,295
roller	53,540	603	knox	72,188	1,177
maven	80,161	1,061	stratos	119,243	1,506
commons-cli	6,859	54	phoenix	273,435	2,134
wss4j	109,259	782	commons-math	186,584	1,685
pdfbox	136,997	1,337	tomcat80	317,555	3,425
aries	181,779	2,710	nifi	354,044	3,954
jmeter	124,358	1,408	vxquery	45,369	751
maven-surefire	58,107	1248	zeppelin	81,218	982
commons-validator	15,930	159	polygene-java	159,748	4,500
stanbol	160,713	1,875	groovy	168,705	2,099
sqoop	76,273	837	apex-core	73,029	1,086
flume	84,882	1,000	apex-malhar	166,972	2,682
rampart	24,729	278	brooklyn-library	40,387	629
kafka	120,995	1,644	beam	199,476	3,631
giraph	97,952	1,870	tomcat85	306,473	3,397
oozie	159,043	1,325	incubator-hivemall	51,984	666
tomcat	303,901	3,428	qpid-proton-j	38,055	613

randomly selected Java projects from the ecosystem[4]. These represent more than a quarter of the Java projects in Apache. Table 3 presents the analyzed systems together with statistics about their magnitude.

We used the Apache Software Foundation Index[5] in order to randomly select the projects that we analyzed. We used three inclusion criteria in order to decide whether we should analyze a project or not. We chose projects in which the main programming language is Java, have at least two years of evolution and are still active at the beginning of 2017. All the analyzed projects use git as version control system and they are hosted on GitHub, whence we cloned them.

The range in terms of weeks of evolution spans from 127 weeks to 767 weeks. We chose to analyze the last 5 years (260 weeks) of the evolution of the projects. The range of the number of classes for the first analyzed commit is from 0 to 7,040 and for the last analyzed commit from 35 - 8,295. At the same time the NCLOC for the first commit ranges from 0 to 450,186 and for the last commit from 5,197 to 635,020.

4 Results and Discussion

This section reports the analysis of the results achieved in our study and aims at answering the four research questions formulated in Sect. 3. A replication kit is available online at https://github.com/digeo/evolution-of-td-in-apache.

RQ_1: How does the technical debt of the open-source systems in the Apache ecosystem evolve over time? To answer RQ_1, for each project in the analyzed ones we created a weekly time series with the accumulation of technical debt. For each series we performed the Mann-Kendall test. The purpose of the Mann-Kendall (MK) test is to statistically assess if there is a monotonic upward or downward trend of the variable of interest over time. A monotonic upward (downward) trend means that the variable consistently increases (decreases) through time, but the trend may or may not be linear.

The null hypothesis (H_0) is that there is no monotonic trend and the alternative hypothesis (H_a) is that a monotonic trend is present. The value of the significance level (alpha error rate) is 0.01 (a = 0.01).

We run MK test for each one of the analyzed systems. Figure 3 visually summarizes the results by presenting the Z values for the analyzed systems. If the Z value is above (below) the horizontal grey line, it indicates that an increasing (decreasing) trend is present.

The Fig. 3 shows that in most of the projects, there is a monotonic upward trend of the technical debt over time.

RQ_2: How does the normalized technical debt of the open-source systems in the Apache ecosystem evolve over time? To address RQ_2 we

[4] The ecosystem contains projects written in more than 20 languages, but the majority of the projects is written in Java.

[5] https://projects.apache.org/projects.html?language#Java.

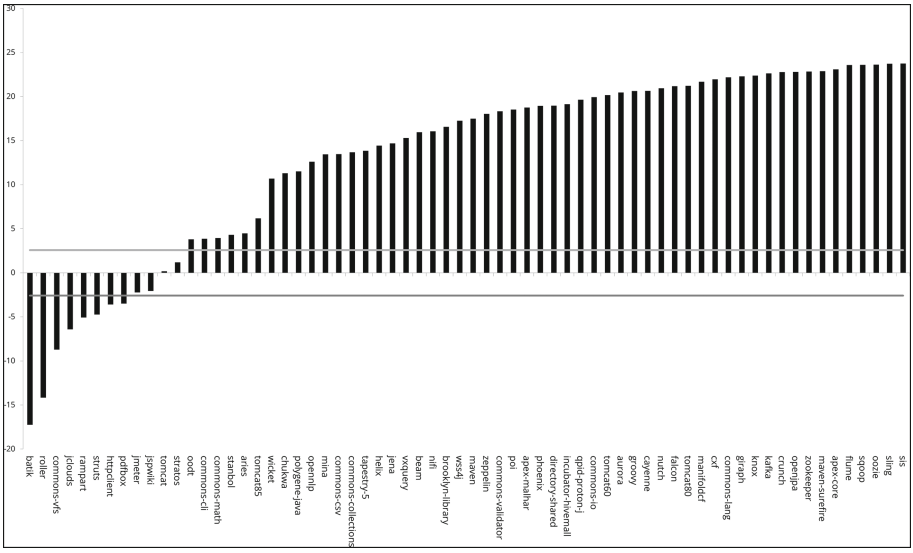

Fig. 3. Trend results for technical debt

extracted two series for each project. The first one contains data for the accumulation of technical debt and the second the number of the lines of code. Then, we divided the two series, namely: the technical debt series with the lines of code series to obtain what we call the *normalized technical debt* time series.

Finally, for each normalized technical debt series we performed again the Mann-Kendall test. Figure 4 presents the Z values for the analyzed systems using the same conventions as before. It shows that:

1. For seven systems (approx. 10%), there is no clear trend (values between the two grey lines)
2. For eleven systems (approx. 20%), the normalized technical debt increases with time (values above the top grey line)
3. for the majority of the systems, the normalized technical debt decreases over time (values below the low grey line)

We find the third result from above encouraging. Indeed, one possible explanation is that the developers of these systems are concerned with paying back the technical debt. This is plausible considering that the systems under analysis are some of the most successful open-source systems and are regarded as high quality projects by the open-source community.

However, another possible explanation could be related to the different phases through which a system evolves; as the system moves towards the maintenance phase, the changes to the system will tend to be smaller such as patches and bug fixes, and thus, less likely to introduce technical debt.

Based on the answer to this research question, we realize that Apache Sling, the system we discussed earlier, was not special in the fact that its normalized

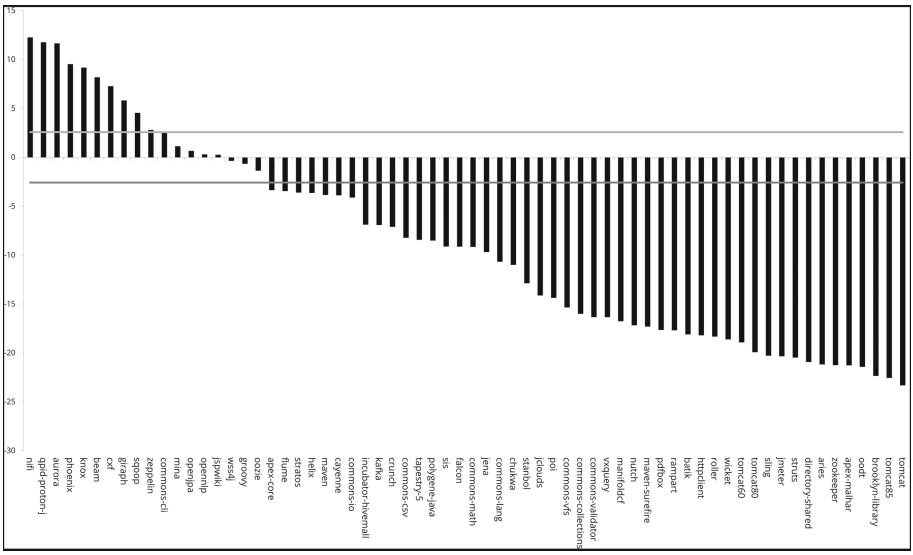

Fig. 4. Trend results for normalized technical debt

technical debt was decreasing. But this is not surprising now, since we see that this is the case with the majority of the systems in the ecosystem, and we have picked Sling at random.

RQ_3: What are the most frequent types of technical debt in the studied ecosystem? To answer this question we summed up all open issues from all the analyzed systems. We only look at the issues that are still open in the last analyzed commit.

To gather insight into the distribution of the various types of issues across the different systems, we use the Gini coefficient. The Gini coefficient is a statistical measure of the degree of variation or inequality represented in a set of values, used especially in analyzing income inequality. Its value is between 0 and 1. A low coefficient is indicative for a uniform distribution in the analyzed values, while a high coefficient is indicative of a very skewed distribution.

Since there are more than a hundred types of issues we do not to present all of them but we limit the presentation to the top ten most frequent ones. The replication kit available online contains the full table of issues in the ecosystem.

Table 4 shows the ten most frequent types of issues encountered in the projects that we analyzed. By analyzing the information in the table, we observe that:

- The top 10 most frequent rule violations account for more than 40% of the issues in the systems. This hints at the fact that, if automated tool support would be developed for these issues, that would make a big difference.
- The most frequent issue is also the most easily remedied, since all the modern IDEs provide an Extract Constant/Variable refactoring. In fact, a recent

study showed that Extract Constant/Variable is one of the most popular refactoring developers actually in practice [7]. This hints at the possibility that developers are not aware of the literal duplication and that future IDEs could auto-detect and suggest the removal of the problem.
– If we add up the two rules in the list that refer to exception handling, they are more frequent than the most frequent issue. This is a sign that exception handling in Java is still not being approached with sufficient discipline. Also this is a much higher-level abstraction than some of the other frequent issues.
– Code duplication, is another type of high-level, potentially architectural problem. It is not very frequent (2.4% of the issues pertain to it) but it has a very low Gini index, which means, it is very equally distributed among the analyzed systems.

Table 4. The ten most frequent types of technical debt in the Apache ecosystem

#	Issue	Count		Gini
1	String literals should not be duplicated	48,474	(7.0%)	.31
2	The members of an interface declaration or class should appear in a pre-defined order	38,756	(5.6%)	.43
3	Exception handlers should preserve the original exceptions	33,467	(4.8%)	.38
4	The diamond operator ("`<>`") should be used	30,659	(4.4%)	.55
5	Generic exceptions should never be thrown	29,393	(4.2%)	.47
6	Statements should be on separate lines	25,674	(3.7%)	.73
7	Control flow statements "`if`", "`for`", "`while`", "`switch`" and "`try`" should not be nested too deeply	24,513	(3.5%)	.34
8	Sections of code should not be "commented out"	22,039	(3.2%)	.52
9	Source files should not have any duplicated blocks	16,456	(2.4%)	.22
10	"`@Override`" should be used on overriding and implementing methods	16,291	(2.4%)	.64

RQ_4: What are the most costly to fix types of technical debt in the studied ecosystem? To answer this question we summed up all the open issues from all the analyzed systems but this time, we looked at the effort instead of the frequency. We are still only looking at the issues that are still open in the last analyzed commit.

Table 5 shows the ten most expensive in terms of effort types of issues in the analyzed projects. By analyzing the information in the table, we observe that:

– Code duplication, is the most expensive to fix in terms of the estimated required time. The function for estimating the time required to remove duplication estimates the effort linearly with the cardinality of the clone.

– Just as with the frequency, exception handling is again the most time-consuming problem to fix. The two types of issues regarding exceptions, account together for more than 13% of the estimated time for paying back the technical debt.
– Rule 3, is responsible in the ecosystem for 8.4% of the effort. Compared with the Apache Sling system presented in the Motivating Example section which had 13% this is much lower. This would probably be useful information for the developers of Sling.

Table 5. The ten most costly to fix types of technical debt in the Apache ecosystem

#	Issue	Effort in minutes		Gini
1	Source files should not have any duplicated blocks	967,490	(13.8%)	.33
2	String literals should not be duplicated	642,122	(9.2%)	.36
3	Generic exceptions should never be thrown	587,860	(8.4%)	.47
4	Cognitive Complexity of methods should not be too high	353,527	(5.0%)	.37
5	Exception handlers should preserve the original exceptions	334,670	(4.8%)	.39
6	Methods should not be too complex	257,213	(3.7%)	.34
7	Control flow statements "`if`", "`for`", "`while`", "`switch`" and "`try`" should not be nested too deeply	245,130	(3.5%)	.34
8	The members of an interface declaration or class should appear in a pre-defined order	193,780	(2.8%)	.43
9	Dead stores should be removed	165,990	(2.4%)	.42
10	Standard outputs should not be used directly to log anything	154,390	(2.2%)	.52

Since we cannot present all the 232 issues uniquely detected by the tool, we summarize their magnitude by computing again the Gini coefficient for the estimated effort per issue. Summing up the percentage of all the issues in Table 5 shows that 55.8% of all the estimated effort is due to these ten issues. We conjecture that if progress was made towards eradicating some of top problematic issues, the community would make considerable progress in avoiding much technical debt.

5 Threats to Validity

In this section, we present and discuss possible threats to the validity of our study.

Construct validity reflects how far the studied phenomenon is connected to the intended studied objectives. The main threats related to construct validity

are due to possible inaccuracy in the identification of technical debt. Since we relied on the default SonarQube rules and the default threshold for each rule in order to detect the violations leading to technical debt, the results are subject to the SQALE model assumptions. This threat is partially mitigated by the fact that the analysis of technical debt evolution implies a relative rather than an absolute assessment of technical debt for the examined systems.

Since the Research Questions have been investigated through a case study, threats to the reliability should be examined. Reliability is linked to whether the experiment is conducted and presented in such a way that others can replicate it with the same results. We believe that the documentation of the adopted research process along with the online replication kit will facilitate any researcher who is interested in replicating this study.

Finally, as in any case study, external validity threats apply, limiting the ability to generalize the findings derived from sample to the entire population. However, the sixty-six systems that we analyzed have been randomly selected from the Apache ecosystem, and represent above a quarter and below a third of all contained Java projects. Moreover, we do not claim that the results on TD evolution or the types of TD hold for other Java systems or different ecosystems.

6 Related Work

This section reports the studies that are related to our work. Specifically, we report report empirical studies that study the Apache ecosystem, studies that deal with the introduction, evolution, and the survivability of the code smells on OSS projects and finally, studies that study the impact of code smells on the OSS projects.

Evolving code smells and software metrics. The evolution of code smells has been studied extensively. One of the first studies is by Olbrich et al. [8], who investigated the evolution of two code smells, namely God Class and Shotgun Surgery. They analyzed historical data of two large OSS projects from the Apache foundation: Apache Lucene and Apache Xerces 2 J, the results of their study report (i) that during the evolution of the projects there are phases that the number of these code smells decreases and phases that this number increases and (ii) the size of the system does not affect these changes.

Zazworka et al. [9] conducted a case study on the design debt. They analyzed two sample applications by a software development company and they investigated how God Classes affect the maintainability and the correctness of the projects. The results of their study show that God Classes have higher change-proneness when they are compared to the non-God Classes. Furthermore, they suggest that God Classes be seen as instances of technical debt and also they point that if the developers split the God Classes into multiple smaller classes that could lead to the generation of more problematic classes and that would have as result an increment to the number of the files that has to be edited.

Peters and Zaidman [10] also conducted a case study on the lifespan of the following code smells: God Class, Feature Envy, Data Class, Message Chain

Class, and Long Parameter List Class. They mined open-source projects and their main finding reports that the engineers are aware of the existence of the code smells in their systems but they do not worry for their impact and that has as result to perform very few refactoring activities. The main deference between our study and their that they analyzed only a small number of Java projects (only seven) and they focused their study only on five code smells. Furthermore, they did not measure how much effort is required in order to remove them.

Chatzigeorgiou and Manakos conducted a study on the evolution of code smells and they report that as the projects evolve over time the number of code smells increases [11]. Furthermore, the developers of the projects perform very few actions in order to remove the code smells from the projects.

All the previous studies focused on a limited number of types of smells and small number of systems. In contrast, Curtis et al. [12] performed a large-scale study on many business applications. They used more than 1200 rules of good architectural and coding practice and they reported the TD of 745 business applications. The main difference between their study and ours is the focus: we focused only on Java OSS projects by the Apache Foundation, they analyzed a big number of business applications that have been developed on many languages as diverse as COBOL, C++, .NET, ABAP, and Java. The similarity between their study and ours is that we also used a set of good architectural and coding practices.

Software Ecosystems. Software ecosystems have been studied in many contexts: their evolving size and developer activity [13,14], their evolving dependencies [15,16], their API evolution [17]. The very ecosystem that we study in this paper, Apache, has been studied from the perspective of sentiment analysis on the mailing lists [18] and the evolution of dependencies between the projects in an ecosystem [15].

One study on open-source systems that comes close to ours in its focus is the one of Tufano et al. [19] who contacted an empirical study on 200 OSS projects. They analyzed projects from three ecosystems namely Apache, Android, and Eclipse and they investigated questions about code smell life cycle. They found that the most code smells are introduced with the creation of the class or file when the developers implement new features or enhance already exist ones. They also report that the majority of the smells are not removed during the project's evolution and few are removed as a direct consequence of refactoring operations. Our study differs from their work in that we focus on the trends at the system level, and we also consider the estimated time that is required in order to resolve the issues of a project.

7 Conclusions and Future Work

In this paper we have studied sixty-six Java systems from the Apache ecosystem. We analyzed cumulatively more than 16,000 weekly commits and we mined 695,731 project issues as reported by SonarQube. From this data we have learned

that in the majority of the systems that we studied, there is a significant increase trend on the size, number of issues, and on the complexity metrics of the project. On the other hand, the normalized technical debt decreases as the project evolves.

Some of the most frequently occurring issues regard low-level coding problems some of which could probably be decreased with good IDE support (e.g. duplicated strings). On the other hand, the most expensive types of technical debt that must be paid back in the ecosystem are actually higher-level problems: duplicated code and ad-hoc exception handling. Exception mis-handling is more unevenly distributed in the ecosystem than code duplication.

One of the reasons for which this study did not analyze the entire Apache ecosystem but rather a sample of it is the slowness of the analysis using SonarQube for which the computation time required is linear with the number of versions. In order to allow the analysis of more systems and also a finer temporal granularity level than a week, in the future we will investigate approaches that would provide better scalability.

In the paper we also observed that a very small minority of problem types is responsible for the vast majority of estimated technical debt. We conjectured that if progress was possible towards *preventing* some of the top problematic issues the community could *avoid incurring* a large percentage of the technical debt in the first place. Even if for other communities the problem ranking would be different, we believe that the approach of aggregating the information from system level to the entire ecosystem will always provide valuable insights. Indeed we consider this to be one of the take-away messages of this study.

Finally, although larger than many earlier studies on the evolution of technical debt in open-source systems, our study is still limited to a random sample from one ecosystem. It would be valuable if these results would be replicated by other researchers in other open-source ecosystems, and maybe also in other languages.

References

1. Li, Z., Avgeriou, P., Liang, P.: A systematic mapping study on technical debt and its management. J. Syst. Softw. **101**(C), 193–220 (2015). http://dx.doi.org/10.1016/j.jss.2014.12.027
2. Manikas, K.: Revisiting software ecosystems research. J. Syst. Softw. **117**(C), 84–103 (2016). http://dx.doi.org/10.1016/j.jss.2016.02.003
3. Lungu, M.: Reverse engineering software ecosystems. Ph.D. dissertation, University of Lugano, November 2009. http://scg.unibe.ch/archive/papers/Lung09b.pdf
4. Campbell, G.A., Papapetrou, P.P.: SonarQube in Action, 1st edn. Manning Publications Co., Greenwich (2013)
5. Ilkiewicz, M., Letouzey, J.-L.: Managing technical debt with the sqale method. IEEE Softw. **29**, 44–51 (2012)
6. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. SIGSOFT Softw. Eng. Notes **17**(4), 40–52 (1992)

7. Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. In: Proceedings of the 31st International Conference on Software Engineering, ICSE 2009, pp. 287–297. IEEE Computer Society, Washington, DC (2009). http://dx.doi.org/10.1109/ICSE.2009.5070529
8. 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, October 2009
9. Zazworka, N., Shaw, M.A., Shull, F., Seaman, C.: Investigating the impact of design debt on software quality. In: Proceedings of the 2nd Workshop on Managing Technical Debt, MTD 2011, pp. 17–23. ACM, New York (2011). http://doi.acm.org/10.1145/1985362.1985366
10. 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, March 2012
11. Chatzigeorgiou, A., Manakos, A.: Investigating the evolution of code smells in object-oriented systems. Innov. Syst. Softw. Eng. **10**(1), 3–18 (2014). http://dx.doi.org/10.1007/s11334-013-0205-z
12. Curtis, B., Sappidi, J., Szynkarski, A.: Estimating the size, cost, and types of technical debt. In: Proceedings of the Third International Workshop on Managing Technical Debt, MTD 2012, pp. 49–53. IEEE Press, Piscataway (2012). http://dl.acm.org/citation.cfm?id=2666036.2666045
13. Vasilescu, B., Serebrenik, A., Goeminne, M., Mens, T.: On the variation and specialisation of workload - a case study of the gnome ecosystem community. Empirical Softw. Eng. **19**(4), 955–1008 (2013)
14. Lungu, M., Malnati, J., Lanza, M.: Visualizing gnome with the small project observatory. In: Godfrey, M.W., Whitehead, J. (eds.) MSR, pp. 103–106. IEEE Computer Society (2009). http://dblp.uni-trier.de/db/conf/msr/msr2009.html#LunguML09
15. Bavota, G., Canfora, G., Penta, M.D., Oliveto, R., Panichella, S.: The evolution of project inter-dependencies in a software ecosystem: the case of apache. In: Proceedings of the 2013 IEEE International Conference on Software Maintenance, ICSM 2013, pp. 280–289. IEEE Computer Society, Washington, DC (2013). http://dx.doi.org/10.1109/ICSM.2013.39
16. Decan, A., Mens, T., Claes, M., Grosjean, P.: When github meets cran: an analysis of inter-repository package dependency problems. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 1, pp. 493–504, March 2016
17. Robbes, R., Lungu, M., Roethlisberger, D.: How do developers react to API deprecation? the case of a Smalltalk ecosystem. In: Proceedings of the 20th International Symposium on the Foundations of Software Engineering (FSE 2012), pp. 56:1–56:11 (2012)
18. Tourani, P., Jiang, Y., Adams, B.: Monitoring sentiment in open source mailing lists: exploratory study on the apache ecosystem. In: Proceedings of 24th Annual International Conference on Computer Science and Software Engineering, CASCON 2014, pp. 34–44 (2014)
19. 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. In: Proceedings of the 37th International Conference on Software Engineering, ICSE 2015, vol. 1, pp. 403–414. IEEE Press, Piscataway (2015). http://dl.acm.org/citation.cfm?id=2818754.2818805