How Do Developers Fix Issues and Pay Back Technical Debt in the Apache Ecosystem?

Georgios Digkas, Mircea Lungu, Paris Avgeriou
Johann Bernoulli Institute for Mathematics and Computer Science
University of Groningen
Nijenborgh 9, 9747 AG, Groningen, The Netherlands
g.digkas@rug.nl, m.f.lungu@rug.nl, paris@cs.rug.nl

Alexander Chatzigeorgiou, Apostolos Ampatzoglou
Department of Applied Informatics
University of Macedonia
Egnatia 156, 546 36, Thessaloniki, Greece
achat@uom.gr, apostolos.ampatzoglou@gmail.com

***Abstract*—During software evolution technical debt (TD) follows a constant ebb and flow, being incurred and paid back, sometimes in the same day and sometimes ten years later. There have been several studies in the literature investigating how technical debt in source code accumulates during time and the consequences of this accumulation for software maintenance. However, to the best of our knowledge there are no large scale studies that focus on the types of issues that are fixed and the amount of TD that is paid back during software evolution. In this paper we present the results of a case study, in which we analyzed the evolution of fifty-seven Java open-source software projects by the Apache Software Foundation at the temporal granularity level of weekly snapshots. In particular, we focus on the amount of technical debt that is paid back and the types of issues that are fixed. The findings reveal that a small subset of all issue types is responsible for the largest percentage of TD repayment and thus, targeting particular violations the development team can achieve higher benefits.**

***Index Terms*—Software Evolution, Technical Debt, Mining Software Repositories, Empirical Study, Apache Software Foundation**

I. Introduction

Technical Debt (TD) is a powerful metaphor that represents shortcuts taken in a software development project, usually to meet business goals such as limited time or budget [1]. Technical debt concerns mostly invisible parts of the system (i.e. no visible features or defects) that have a detrimental effect on the maintainability and evolvability of software [1]. Technical Debt cannot realistically be eliminated but it does need to be proactively managed in order to be kept at a sustainable level [2].

One of the primary activities of Technical Debt Management is termed Technical Debt Monitoring [2]: the activity of continuously inspecting the levels of TD throughout time. Monitoring technical debt aims to identify trends in the evolution of TD and alert the development teams in case certain TD items increase beyond a threshold, or even worse grow out of control. In that sense monitoring helps to identify and prioritize repayment actions where TD items are resolved (e.g. through refactoring).

In this paper we study the TD evolution of fifty-seven open-source software projects by the Apache ecosystem[1][2], through a longitudinal case study, with the aim of identifying cases in which repayment activities take place. Although in the literature there are many studies which investigate projects by the Apache ecosystem (since they are highly appreciated and used by many software engineers), in this work we narrow down our scope to the types of issues that are fixed and the amount of TD that is paid back over time.

Structure of the Paper: The rest of the paper is organized as follows: In Section II we discuss related work on the evolution of software projects with respect to their quality and code smells in particular. Section III clarifies methodological issues related to the collection of data for the repayment of TD. The protocol of the case study is outlined in Section IV, while the Results are presented and discussed in Section V. Key observations and open questions resulting from the study are outlined in Section VI. Threats to the validity of the study are discussed in Section VII. Section VIII provides information regarding the replication package for the study. Finally, we conclude in Section IX.

II. Related Work

This section reports studies that are related to our work. These include empirical studies that investigate the evolution of open-source software projects, the evolution of TD over time and studies that deal with the survivability and impact of code smells in open-source software projects.

Bavota et al. [5] have studied the evolution of inter-dependencies of 147 Java projects over a period of 14 years that belong in the Apache ecosystem. The results of their study show that there is a linear increase trend in the number of projects but the number of dependencies between them tend to increase exponentially. They have also found that the developers do not upgrade immediately the dependencies of their projects unless it is a major release.

[1]Apache Software Foundation is one of the biggest communities which "provide software products for the public good"

[2]In this paper we consider an ecosystem to be *"a collection of software systems that are developed and co-evolve in a shared environment"* [3] even if other definitions exist [4]

One of the first studies that investigates the evolution of code smells in the literature is by Olbrich et al. [6]. They analyze historical data of two projects by the Apache Foundation, namely Apache Lucene and Apache Xerces 2 J and they study how the God Class and Shotgun Surgery smells evolve over time. The main findings of their study are that during the evolution of these projects, there are phases where the number of these smells decreases and phases where the number increases. Moreover, these phases are not affected by the size of the systems.

Peters and Zaidman [7] developed a tool which is called SACSEA. The tool computes the lifespans of the following code smells: God Class, Feature Envy, Data Class, Message Chain Class, and Long Parameter List. The tool has been employed to analyze eight open-source software projects. The main finding of their study reports that software engineers are aware of the existence of the code smells in their projects. However, there is evidence that they perform very few refactoring activities.

Chatzigeorgiou and Manakos [8] have also investigated the evolution of code smells in object-oriented projects. They studied the evolution of Long Method, Feature Envy, State Checking, and God Class smells throughout successive versions of two open-source software projects, namely: JFlex and JFreeChart. The results of their study show that as the projects evolve over time the number of code smells increases. Furthermore, the developers do not perform refactorings in order to solve these code smells and in the vast majority of the cases if one code smell is removed, it is probably a side effect of regular maintenance and not a result of intentional refactoring activity.

Zazworka et al. [9] investigate how God Classes impact maintainability and correctness. To this end, they analyzed two sample applications by a software development company. Their findings show that God Classes are more change-prone and there are some cases where they also are more defect-prone, when they are compared to the non-God Classes.

Tufano et al. [10] conducted a study on 200 open-source software projects from three different software ecosystems, namely: Apache, Eclipse, and Android. The main aim of their study is to understand when and why code smells are introduced into the projects and what is their life cycle. In order to answer this question, they focus on the following code smells: Blob Class, Class Data Should be Private, Complex Class, Functional Decomposition, Spaghetti Code. The results of their study, show that a) in the majority of the cases, the code smells are introduced into the projects with the creation of classes or files, b) also, while the project evolves over time, the smelly code artifacts will become more smelly, c) software engineers introduce new code smells when they implement new features or when extend the ones that already exist, d) the developers who introduce new code smells into the projects, 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.

TABLE I
RELATED WORKS

Authors	Year	Projects	Issues
Bavota et al. [5]	2013	147	-
Zazworka et al. [9]	2011	2	1
Olbrich et al. [6]	2009	2	2
Chatzigeorgiou and Manakos [8]	2014	2	4
Peters and Zaidman [7]	2012	8	5
Tufano et al. [10]	2015	200	5
Digkas et al. [11]	2017	66	232
Maldonado et al. [13]	2017	5	-
Curtis et al. [12]	2012	745	> 1200

Digkas et al. [11] have also studied the evolution of open-source software projects by the Apache Foundation. They analyze the evolution of 66 Java projects, over a period of 5 years, using SonarQube in order to investigate how TD evolves and what are the types of issues that incur it. In their study, they use the default SonarQube settings and they examine 232 different types of issues that incur technical debt. The results of their study show that on the one hand, there is a significant increasing trend on the size, complexity, number of issues, and the technical debt over time. But on the other hand, when the TD is normalized to the size of the project, it decreases over time. This means that the developers either insert better quality code on the projects or they fix some of the open issues.

A large-scale study on 745 business applications has been performed by Curtis et al. [12]. The projects that they analyzed are from 160 companies that belong in 10 industry segments. The analysis that they perform is similar to that of the present study. They used more than 1200 rules of good architectural and coding practices in order to evaluate the quality of these applications. One key difference between our and their study is that we investigate the number of the issues that are fixed and the amount of TD that is paid back.

Maldonado et al. [13] have conducted an empirical study on the removal of Self-Admitted Technical Debt (SATD) by analyzing the code comments of five open-source software by the Apache Software Foundation. The results of their study show that the removal of SATD issues could take from several months and up to a decade. The authors also found that in most of the cases the developer who introduced an SATD issue is the one who removes it when they fix bugs or add new features.

Table I summarizes all studies reported in this section, including the year of publication, the number of analyzed projects and the number of examined issues.

III. STUDYING TECHNICAL DEBT EVOLUTION

In order to study the repayment of debt one must study its evolution, and in order to study the evolution of debt one must be able to detect it in the first place. Different types of debt can be detected from various sources of information: source code, architectural documentation, issue trackers, mailing lists, etc. However, to detect the evolution of debt over a long period of time and in a large number of systems, restricting the study of debt to that which can be inferred based on the study of source code makes the challenge more approachable.

A. Using SonarQube for Debt Estimation

In this study, we use SonarQube [14] to detect fixes of rule violations and in this way we compute the amount of TD that is paid back. SonarQube, relies on a set of rules in order to calculate the amount of TD. During the analysis of any project, SonarQube creates a new *issue* every time a piece of code breaks one of the predefined *rules*. For every issue it assigns an estimate of how much time will be required for someone to solve it. A downside of this decision is the restriction of the study to the particular types of TD that can be detected by source code analysis and in particular by the SonarQube tool platform.

We have decided to use SonarQube because it is one of the most popular tools that are used in industry for the purpose of measuring and estimating debt. Moreover, SonarQube can detect a large number of source code related types of TD.

The tool uses different strategies for estimating TD for different types of issues:

1) **Constant Time** – some issue types are assigned a constant time estimate for their fixing (e.g. "Sections of code should not be commented out" is assigned a constant fixing time independent of the size of the commented out code)
2) **Issue Dependent Time** – some types of issues are estimated to take a time which takes into account their particular characteristics (e.g. cloning takes into account the number of clones)

Finally, to compute the total amount of technical debt per system at a given time, SonarQube calculates the estimated fixing time for all the open issues.

B. Studying Multiple Revision of a System

SonarQube, and other similar tools, cannot take as input *deltas* between system revisions, but instead they parse the entire source code of a system and build a model for every new version submitted for analysis. This means that even if a single line change in a single file will result in a massive and time-consuming analysis step which for large systems can take several minutes.

To study the evolution of a large number of systems one must thus *discretize* their evolution by analyzing only a limited number of revisions of the system. However, although the analysis of multiple revisions is slow, the individual *issues* together with their cost estimation being the most basic unit of debt are identified by the tool across revisions and their identity can be preserved.

The preservation of issue identity is achieved even in the presence of different types of changes. However, one of the situations in which the identity of an issue is lost is whenever a file is renamed and at the same time the content of that file is changed too. On the other hand, if only one of these changes occurs (i.e. renaming of the file or change of content) the tool is still able to preserve the identity of the issues.

C. Detecting Fixed Issues

When analyzing multiple revisions of a system, SonarQube tracks the issues that are fixed and the debt that is repaid. When an issue disappears, it is considered **fixed**.

By investigating manually a series of such fixed issues we observed three types of situations in which the tool considers an issue to be fixed:

1) Source code deletion. If a piece of code that contains some issues that incur TD is deleted, then the TD is paid back automatically. Deleting a piece of code (e.g. a statement of a block) represents TD repayment which can be either intentional or unintentional.
2) Refactoring of the code aiming explicitly at the removal of the issues. If one section of code has been detected to contain TD issues, the refactoring of the corresponding code to remove the issue will result in the repayment of TD.
3) A file-rename-con-file-content-change situation, as described earlier. In this problematic situation, the tool considers all the issues in the old file to be removed and a series of equivalent issues in the new file to have appeared.

Since the goal of this study is the detection of fixed issues and thus, repaid debt, we must be wary of the issues which are fixed in the third situation since they are not actually *fixed* and the corresponding debt is not being *repaid*.

By using the SonarQube API a filter can be implemented to remove the issues in the third category from the list of fixed issues[3]. By using such a filter, in the remainder of this paper, we only talk about the first two types of closed issues.

D. Classifying Issues

SonarQube classifies the issue types into two main categories: based on their type and severity.

Issue Types: If an issue is related to a piece of code that is demonstrably wrong is classified as **bug**. If a piece of code could be exploited by a third person and that could be the reason to harm the system then it is classified as **vulnerability** and finally, as **code smells** are classified all the issues that represent an instance of improper code and they are neither a bug nor a vulnerability.

Issue Severity Levels: In terms of severity, there are five categories namely: blocker, critical, major, minor, and info. Based on the main object of their impact, there are the following three sub-categories:

1) *Impact on the system.* **Blocker** and **critical** designation expresses the impact that an issue can have in the behavior of the application in production. Blocker issues have high probability to negatively impact the system, which is a reason that the software engineers should fix them as soon as possible. Critical issues have lower probability, when we compare to the blocker, to impact negatively the system.

[3]Code for the filter is available in the online replication package for the paper

2) *Impact on the productivity of a software engineer*. **Major** can have high impact and **minor** have little impact.
3) *Everything else*. The last category is the **info**. This category collects all the issues that are neither a bug nor a quality flaw.

IV. Study Design

The goal of our study is to analyze the evolution of open-source software projects in the Apache ecosystem for the purpose of understanding and investigating the types of the issues that have been fixed and the amount of TD that is paid back. More specifically, our study aims at addressing the following five research questions (RQs):

1) *How does the issue fixing rate vary for different projects?* The goal of this research question is to investigate whether software development practices differ with respect to TD repayment among projects.
2) *What is the fixing prevalence of the various issue types?* This RQ aims at identifying which issues are fixed more often than others.
3) *How does the fixing rate vary for different issue types?* Because the number of fixes is dependent on the number of issues for each particular type, this RQ investigates the fixing rate, i.e. the percentage of fixes over the total number issues present in each system.
4) *How is the effort of paying back TD distributed across the various types?* Since different issue types demand different amounts of effort to resolve them, this RQ sheds light into the issues which have yielded the higher benefit in terms of TD repayment.
5) *After how much time is TD paid back?* Similarly to previous studies we analyze the time that is required to fix issues, for the types that have been identified as more beneficial in terms of TD repayment.

A. Project Selection

For the purpose of this study, we chose to analyze projects by the Apache Software Foundation. Since the analysis that we performed is computationally intensive, we used the Apache Software Foundation Index[4] in order to randomly select a sample of fifty-seven Java projects.

The three main inclusion criteria that we used are the following:

1) In terms of programming languages, we chose to analyze projects that the **main programming language is Java and have at least 100 classes**
2) In terms of project evolution, we included projects that have been developed for **at least two years and 1000 commits**, and
3) In terms of activity, we included only projects that there was commit activity in 2017, in other words, projects that are **still active in the year when we conducted this study**

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

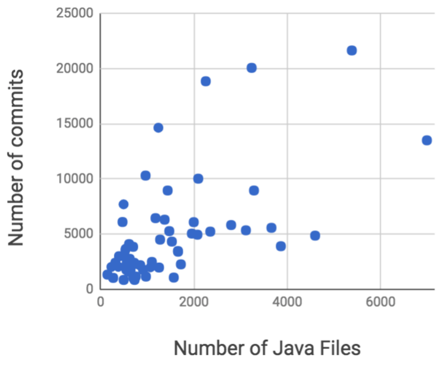

Fig. 1. Correlation between the size of a project (in number of files) and commit density

Figure 1 shows considerable variation in both the number of commits and the size of the systems as measured by counting the number of Java files.

B. Issue Selection

We are limiting the focus of this paper to those issues in the blocker-critical-major category: bugs or flaws that have a probability of affecting the behavior of the application or can highly impact the productivity of developers. The reason for this is that many of the minor and info issues are quite *minor* including:

- "Functions should not be defined with a variable number of arguments" (info)
- "Field names should comply with a naming convention" (minor)
- "Useless imports should be removed" (minor)
- "Unused local variables should be removed" (minor)

Figure 2 illustrates the results of aggregating the issues in the analyzed system based on their severity level. In total 5,041 Blocker, 44,421 Critical, and 105,847 Major issues have been fixed in the full history of the fifty-seven systems under study. The light gray background presents the total number of issues of that kind. The Major have a much higher percentage of fixes than the other types (see Figure 2).

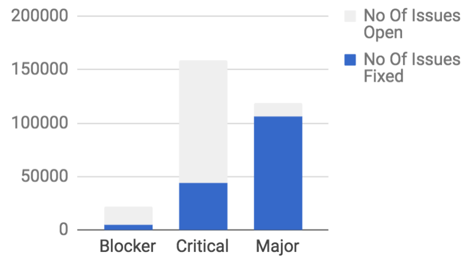

Fig. 2. Distribution of resolved issues by priority type (the light gray background presents the total number of issues of that kind)

This narrowing of the focus results in 160 types of issues to which we will limit our discussion to henceforth.

V. Results and Discussion

A. How does the issue fixing rate vary for different projects? (RQ$_1$)

The goal of this research question is to investigate whether software development practices differ with respect to TD repayment among projects.

Figure 3 shows several types of information for every project. The middle histogram presents the absolute number of issues fixed in a project (dark green) and the number of issues which are still open (light green). The bottom histogram presents in dark green the **issue fixing rate** – the percentage of the issues in that project which are closed[5]. The top part of the Figure 3 presents the size of the corresponding systems

Based on the figure we observe that:

- The highest fixing rate is presented by projects `nutch` and `jmeter` which both have more than 70% of the issues that appeared during their evolution fixed. This could be possibly attributed to the fact that the development team of `nutch`[6] uses SonarQube to evaluate its quality, while `jmeter`[7] is itself a SonarQube plugin.
- Four projects have a fixing rate of less than 10%. They are projects of varying sizes, with `polygene-java` having more than 2K Java files.

[5]The projects are sorted in decreasing order of their issue fixing rate
[6]https://svn.apache.org/repos/asf/nutch/site/publish/sonar.html
[7]https://docs.sonarqube.org/display/SONARQUBE45/JMeter+Plugin

- If we compress further the information about fixing rates as in Figure 4, which shows a histogram depicting the number of projects for several classes of fixing rate with a bin size of 10%, the largest category, with 20 elements, corresponds to a fixing rate of between 20% and 30%.

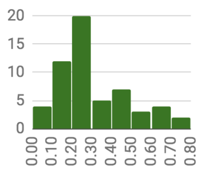

Fig. 4. Distribution of fixing rates in examined projects for bins of size 0.10

- There is no clear correlation between the absolute number of issues found in the system and the percentage of fixes: `batik` (rightmost) and `nifi` (7th from right) have very large numbers of issues but very low fixing percentages while for similar numbers of issues `nutch` (leftmost) and `jmeter` (2nd from left) have very high fixing rates.
- There seems to be no clear correlation between the size of the projects and the fixing rate: `sis` (4th from left) with one of the highest fixing ratios and `batik` (righmost) with the lowest fixing ratio are about the same size.

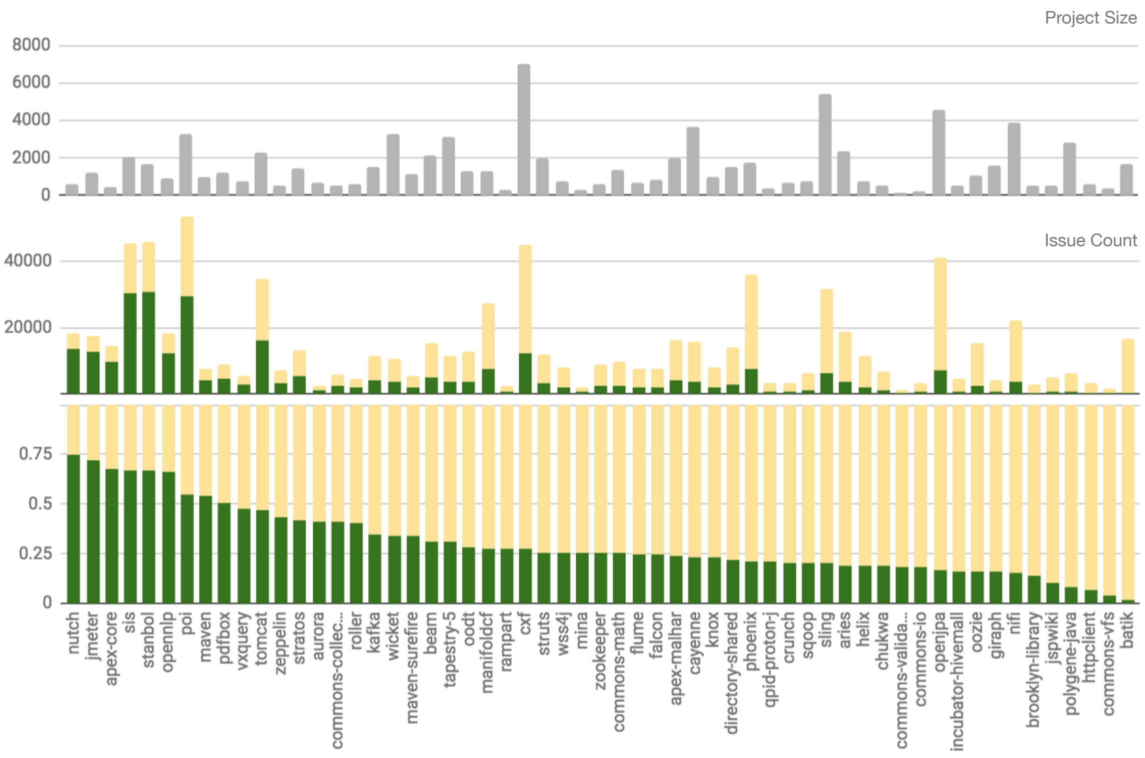

Fig. 3. Number of open (light yellow) and closed (dark green) issues per project. The top part of the figure presents the sizes of the corresponding projects; the middle absolute numbers of open/closed issues; the bottom part presents percentages of open/closed issues

B. *What is the fixing prevalence of the various issue types? (RQ_2)*

To answer RQ_2 we sum up all fixed issues, without differentiating among projects. The histogram of Figure 5 presents the distribution of the number of fixed issues for each type: every vertical bar is an issue type, and its height is proportional to the number of issues of that type. The figure shows a strongly skewed distribution where a handful of issue types are fixed very frequently while for most types fixes are rarely encountered.

Computing the Gini index for the prevalence of the issues we obtain a very high value of 0.845[8]. This means that the "wealth" of issue fixes is very inequally distributed across the issue types.

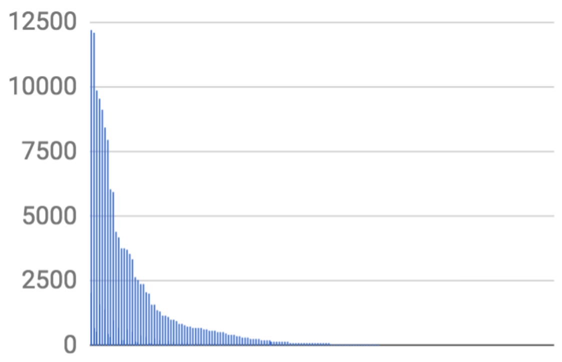

Fig. 5. Distribution of resolved issues by issue type for the 160 types of issues in this study

In order to focus on the left side of Figure 5, Table II shows the ten most frequently fixed types of issues encountered in the analyzed projects. Instead of presenting the absolute number of fixed issues for each issue type, we present the prevalence (column P%) which is, the percentage of that type of issues fixed from the total number of fixes.

TABLE II
MOST FREQUENT TYPES OF RESOLVED CODE SMELLS (TYPES RELATED TO EXCEPTIONS, COMPLEXITY, AND DUPLICATION ARE HIGHLIGHTED)

Issue	P%[a]
1. Statements should be on separate lines	7.8
2. Sections of code should not be "commented out"	7.8
3. String literals should not be **duplicated**	6.4
4. **Exception** handlers must preserve the original exceptions	6.1
5. Control flow statements should not be **nested too deeply**	5.9
6. Cognitive **Complexity** of methods should not be too high	5.4
7. Generic **exceptions** should never be thrown	5.1
8. Standard outputs should not be used directly for logging	3.9
9. Source files should not have any **duplicated** blocks	3.8
10. Synchronized classes Vector, [...] should not be used	2.8

[a] Prevalence percentage

Several high-level observations that can be derived from the table are:

1) **Skewdness**. The ten issue types (out of 160!) in the table account for 55.14% of all the issue fixes in the studied systems. All the fifty-seven systems contain fixes of these issues with the exception of issue #1 missing in five systems and issue #10 absent from 2 systems.
2) **Ease of fix**. The most frequently occurring fixes are also probably some of the easiest to be remedied, as separating statements to different lines (issue #1), removing commented code (issue #2), and removing string literal duplication (issue #3) that normally have no side effects.
3) **Duplication**. Issues related to duplication appear twice in the table (#3 and #9), are found in all the analyzed systems, and account together for more than 10% of the total issues fixed revealing that code clones are identified and handled by developers in numerous instances.
4) **Exception Handling**. Issues related to exception handling also appear twice in the table amounting together for more than ten percent of the total fixes. They also affect all the systems.

Types of Issues: It should be noted that most of the fixed issues are not necessarily language specific; the first language specific issue is at position 10, namely "Synchronized classes Vector, Hashtable, Stack and StringBuffer should not be used".

All the issues in the table are classified by SonarQube as code smells, i.e symptoms that possibly indicate a deeper problem, but are not bugs. The ten most frequently fixed problems that are classified as bugs are shown in Table III.

TABLE III
MOST FREQUENT TYPES OF RESOLVED BUGS AND VULNERABILITIES

Issue	P%[a]
Bugs	
16. Resources should be closed	2.15
22. Null pointers should not be dereferenced	1.30
36. InterruptedException should not be ignored	0.49
41. Floating points should not be tested for equality	0.43
45. Conditionally executed blocks should be reachable	0.38
70. All branches in a conditional structure should not have exactly the same implementation	0.11
72. Identical expressions on 2 sides of binary operator	0.10
78. Assignments should not be redundant	0.08
79. Jumps should not occur in "finally" blocks	0.08
80. Classes should not be compared by name	0.08
Vulnerabilities	
68. SQL binding mechanisms should be used	0.11
76. Credentials should not be hard-coded	0.08
127. File.createTempFile should not be used to create a directory	0.01

[a] Prevalence percentage

Finally, with respect to issues tagged as vulnerabilities, there are only three vulnerability types, in the blocker-critical-major category, that have been detected as fixed. These fixes are responsible for less than 0.20% of the total fixes, as shown in Table III. Although not shown in the table, these issues occur in very few projects, and also have a low fixing rate.

[8]The index varies between 0 for a population with perfect equality and 1 for a population with perfect inequality

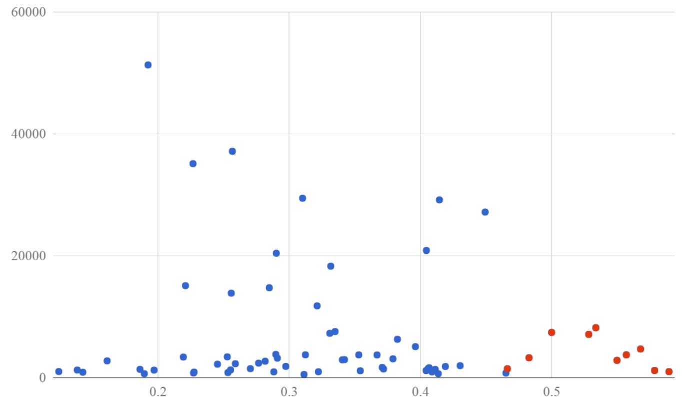

Fig. 6. Fixing rate (horizontal axis) vs. Number of Issues for a given Issue Type

C. *How does the fixing rate vary for different issue types? (RQ_3)*

The reason for asking this research question is understanding whether some types of technical debt are repaid more often than others – and thus, could represent candidates for debt that is more important for the developers of the Apache ecosystem[9].

By computing the fixing rate as the ratio between all the issues of a given type that are closed over the ones which are open we found six issue types with a fixing rate of 100%. However, they are very rare with the most popular one ("Future keywords should not be used as names") occurring only 21 times and the second most popular one ("Variables should not be self-assigned") occurring only 10 times in the whole the history of all the fifty-seven projects[10]. The reduced prevalence of the issues decreases their relevance for a broader developer community. To increase the relevance of the observations, in the remainder of this section, we limit our discussion to issues that have a minimum of 500 discovered occurrences.

Issues with at least 500 occurrences: Figure 6 shows a scatter plot of the 69 issue types which have at least 500 occurrences[11]. The red data points correspond to the ten issue types with the highest fixing rate which are detailed below. The figure shows that:

- The majority of the issue types have a fixing rate between 20% and 50%
- The issue types with the highest fix rates (marked in red) are distinct from the issue types with the highest prevalence in the studied systems.

Table IV presents details about the ten issue types with the highest fixing rate. There are eight issue types which are fixed in at least 50% of the cases. One salient observation is that, in the table, the majority of the issues are Java specific (marked with ☕).

[9]There is no guarantee that if something has a high fixing rate it is necessarily important for the developers; however, we expect that those issues which are important, will be found among those with a high fixing rate. On the other hand, it is safe to say that issue types with low fixing rate, are very likely to not be important for developers

[10]The others are available in the online replication package

[11]By removing the other issue types we are removing a lot of "noise" data points that are crammed on the bottom of the figure along the axes

TABLE IV
ISSUES WITH THE HIGHEST FIXING RATE (☕ARE JAVA-SPECIFIC)

Issue	Count	F[a]
Conditionally executed blocks should be reachable	594	59
☕ Replace Map.get/test with single method call	694	58
☕ Deprecated elements should have both the annotation and the Javadoc tag	2678	57
Unused "private" fields should be removed	2094	56
Boolean expressions should not be gratuitous	1571	55
☕ Synchronized classes Vector, Hashtable, Stack and StringBuffer should not be used	4381	53
☕ Constructors should not be used to instantiate "String" and primitive-wrapper classes	3761	52
Dead stores should be removed	3720	50
☕ @Override should be used on overriding [...]	1586	48
Unused "private" methods should be removed	689	47

[a]Fixing Rate

TABLE V
ISSUE TYPES WITH THE LOWEST FIXING RATE

Issue	Count	F[a]
Floating point numbers must not be tested for equality	665	28
Expressions should not be too complex	597	26
Exception handlers should preserve the original exceptions	9535	26
Methods should not be empty	3549	26
☕ Lambdas and anonymous classes should not have too many lines of code	869	25
Instance methods should not write to "static" fields	547	25
Generic exceptions should never be thrown	7961	23
Resources should be closed	3338	22
Utility classes should not have public constructors	746	22
String literals should not be duplicated	9875	19

[a]Fixing Rate (rounded to the closest integer)

Table V lists the ten issue types with the lowest fixing rate. The table contains fewer Java-specific issues (it could be argued whether there is one or several). Instead it shows generic issues related to best practices which were seen in an earlier discussion to have a high prevalence in the case study: exception handling, complexity, duplication.

D. *How is the effort of paying back TD distributed across the various types? (RQ_4)*

This question aims to shed light into the benefits resulting from paying back TD, as the reduction of the TD principal is not only related to the frequency of fixes but also the relative contribution of each issue to the overall TD.

To answer this research question we have summed up the effort that would have been nominally required to resolve all fixed issues, according to the estimates provided for each issue type by SonarQube. The total amount of TD estimated to be paid back is 30,200 hours and Table VI shows the percentage of repaid TD by issue type. The "Change in ranking" column from the table shows the relative change in the global ranking of a given issue with respect to the prevalence ranking discussed in RQ_2.

TABLE VI
SMELLS WHOSE RESOLUTION HAS YIELDED THE HIGHER BENEFIT

#	Issue	Est. Repaid Debt	Change in Ranking[a]
1	Source files should not have any **duplicated** blocks	11.78	↑8
2	Cognitive **Complexity** of methods should not be too high	11.19	↑4
3	Generic **exceptions** should never be thrown	8.83	↑4
4	String literals should not be **duplicated**	6.64	↓1
5	**Exception** handlers should preserve the original exceptions	5.29	↓1
6	Control flow statements ”if”, ”for”, ”while”, ”switch” and ”try” should not be **nested too deeply**	5.07	↓1
7	Synchronized classes Vector, Hashtable, Stack and StringBuffer should not be used	4.86	↑3
8	Methods should not be too **complex**	3.97	↑3
9	Standard outputs should not be used directly to log anything	3.37	↓1
10	Sections of code should not be ”commented out”	3.36	↓8

[a]refers to changes compared to the frequency of code smells.

Changes in Ranking: By analyzing the “Change in ranking” column we observe that the order of issues changes with respect to the frequency of fixes presented in RQ2. This means that the most frequently fixed issues are not necessarily the ones representing the highest amount of TD paid back. The biggest jump in the ranking is observed for the issue of duplication between source files. The fixes of this type of issue are responsible for ≈12% of the total TD that is paid back. That is because the estimation of TD payback time is proportional to the number of duplicated blocks.

Types of Debt: The table shows that three general classes of debt are represented by more than one issue in the top-10 list and together account for more than half the total repaid debt:

1) **Method Complexity** – accounting for about 24% of the total estimated pay back effort – is detected in three cases: issue types #2, #6, and #8. These issues are related to the understandability of the code. Cognitive Complexity measures the maintainability of the code and indicates how difficult is for someone to understand the control flow of a method and to maintain it. Deep nesting of control flow also increases cyclomatic complexity and reduces understandability and testability of the code. Having a code base clear without commented out sections of code also helps software engineers to understand the functionality easier. Commented out sections of code slow down the development and maintenance speed.
2) **Code Duplication** – accounting for about ≈18% of the total estimated pay back effort – is detected in two situations: at file level and at the level of string literals. Code duplicates appear frequently and cost a lot in many different ways. They make the source code difficult to understand, maintain and extend.
3) **Exception Handling** – accounting for about 14% of the total estimated pay back effort – is represented by two cases: issue types #3 and #5. Using generic exceptions prevent methods from handling more specific, application-generated errors in the appropriate way. Moreover, the results imply that developers care about passing forward the original exceptions’ messages and stack traces.

TABLE VII
BUGS AND VULNERABILITIES WHOSE RESOLUTION HAS YIELDED THE HIGHER BENEFIT

Issue	ERD[a]
Bugs	
17. Null pointers should not be dereferenced	1.12
23. Resources should be closed	0.93
31. InterruptedException should not be ignored	0.64
37. Conditionally executed blocks should be reachable	0.49
55. Jump statements should not occur in ”finally” blocks	0.20
58. Floating point numbers should not be tested for equality	0.18
64. All branches in a conditional structure should not have exactly the same implementation	0.14
77. Blocks should be synchronized on ”private final” fields	0.08
79. Non-thread-safe fields should not be static	0.08
83. Synchronization should not be based on Strings or boxed primitives	0.04
Vulnerabilities	
53. Credentials should not be hard-coded	0.211
56. SQL binding mechanisms should be used	0.195
134. File.createTempFile should not be used to create a directory	0.002

[a]Estimated Repaid Debt

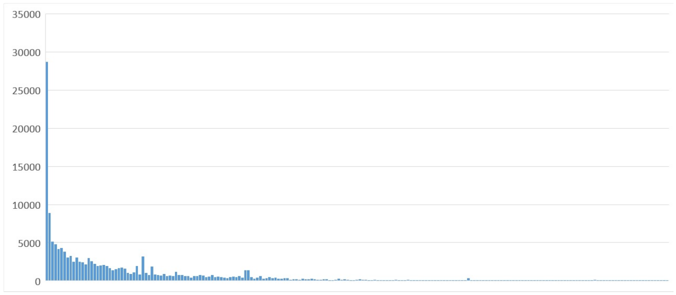

Fig. 7. Histogram of the survival time for the 155K issues in this study with bins of 30 days

E. After how much time is TD paid back? (RQ_5)

This RQ focuses on the survival of the various types of issues that have been identified in the previous research questions focusing on the issues whose resolution has yielded the higher benefit in terms of TD payback. For each one of those issues types, we calculated the number of days that the issue survived, i.e. the difference between the time point at which an issue is introduced to the source code to the time it is resolved[12].

Figure 7 presents the histogram of the fixing time of all the 150K issues found fixed in this study. Taking into account the fact that the total number of fixed issues in this study is slightly more than 155K, the histogram shows that:

- Almost 20% (≈30K/155K) of the issues are fixed within one month of their introduction
- More than 50% of the issues are fixed within the first year
- The lifetime of the issues is a long-tailed distribution with a small number of issues being fixed even after ten years. It seems that one must never lose hope when it comes to the possibility of debt being paid back!

Figure 8 shows the distribution for each of the ten issue types reported in RQ_2 in the form of a boxplot (outliers are not shown in the diagram for simplicity). The figure limits itself to presenting only the respective issue types since, as shown in the previous section, these are responsible with more than half of the effort required for payback.

There is a large difference between the minimum and the maximum duration that is required in order to fix an issue.

The main observations from the survival times of these issues are similar to the corresponding observations about the entire populations. For all issue types in the figure:

- 25% of all instances disappear in less than two months
- 50% of all issue instances disappear approximately within one year from their introduction to the code
- The longest lived issue types exhibit a large variation that spans from 5 years to more than 10 years

Moreover, the figure highlights in similar colors issue types which are similar. One can see that there is a certain similarity in the survivability distribution of the similar issue types.

VI. Observations and Open Questions

Based on this study, we can conclude with several observations about the way technical debt evolved in the Apache Ecosystem over ten years:

[12]One of the limitations of the weekly commits approach is that when we compute the time an issue "survived" we will consider that it was fixed in the last commit of the week even if it was actually fixed earlier that week. In fact, the worst case scenario is when the issue was fixed on the first day of the week, and we only consider it to be fixed in the last day of the week. Thus our results regarding the longevity of a given issue have a built-in maximum imprecision of seven days

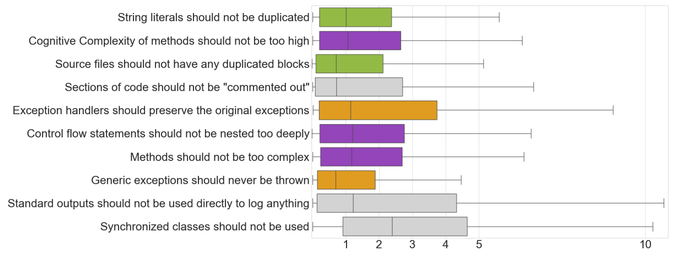

Fig. 8. Distribution of survival time for top the ten issues (according to RQ_2)

- There is a very large variation between the fixing rate of issues that contribute to technical debt among the projects of the same ecosystem
- Only a very small minority of the issue types have a fixing rate of more than 50%. If we consider those types with at least 500 instances, then the count is eight.
- There is a very large variation in the survivability of the different issues: about 10% of the issues are fixed within the first month, about 50% in the first year, and some of the issues can take up to ten years
- Issues related to duplication and exception handling are frequently encountered and rarely fixed by developers

Questions which are still open after this study:

- What percentage of the issue fixing happens during purposeful debt repayment by developers and what percentage happens as a side effect of source code evolution?
- What is the impact of individual developers on issue fixing and technical debt in the studied ecosystem?
- How can we use this information to be able to provide developers with ways of better understand their own system's evolution?
- Do other language ecosystems (Smalltalk, Javascript, R, have already been studied before [15]–[17]) have different attitudes towards debt repayment? How do these results compare against closed-source ecosystems?

VII. Threats To Validity

The results of the study are unavoidably subject to external validity threats since a sample of projects has been analyzed to study how TD is paid back, thus limiting the ability to generalize the conclusions to projects of a different programming language, ecosystem or domain. Nevertheless, the fact that fifty-seven active, large and highly popular projects have been studied partially mitigates threats to generalization.

As explained in the study design, we consider treats an issue as paid back either when the development team intentionally refactored the code or when the code fragment hosting the issue (but not the entire file) has been removed. When the entire file had been deleted, we do not count the contained issues in that file in the removed ones. One threat to the construct validity of the study stems from that fact that issue elimination due to the removal of the corresponding statement, code block or method is consider as paying back TD whereas it could be a side effect of regular maintenance activities. However, even in the presence of this threat the findings on deliberate or unintentional TD payback can be valuable since the focus is on the relative frequency of each issue type rather than the absolute count of issue removals. An additional threat stems from the fact that TD identification is performed on source code artifacts rather than the built version of each project. The lack of build information often leads to false positives. For example, a language specific issue (e.g. the omission of the diamond operator) might be reported for a revision in which the rule does not apply (language version prior to Java 7).

Concerning threats to the repeatability, we believe that this study can be easily replicated by other researchers, since: a) the study protocol is extensively described in this paper, and b) the collection of data and analysis of the results did not rely on any subjective judgment and is based on SonarQube reports.

VIII. Data Set and Replication

As explained before, due to the limitations of multi-version analysis, collecting the data used for this study was time consuming and computationally intensive. In total, the analysis of 19,800 revisions of fifty-seven systems took more than a month even though we used multiple machines for the analysis. We took weekly snapshots for each one of the selected projects and we picked the last commit of each week. Then, for that revision we analyzed all the Java files (including the test classes) in order to detect which of the open issues were fixed and which additional issues were introduced. The resulting PostgreSQL database dump is more than 28GB in size. An online Git repository available at https://github.com/td-evolution-in-apache contains a replication package which includes:

- A series of scripts that can be used for data analysis
- A spreadsheet with all the data that was used in this paper
- Information about how to obtain the full database dump and how to import it a SonarQube instance for those users who would want to explore it using SonarQube's API

We would be happy to support other researchers in using this dataset for their investigations as we believe there are more open questions that this study raises than it answers.

IX. Conclusion and Future Work

The Technical Debt metaphor has been widely adopted and one of its underlying implications is that principal and interest are not regularly paid back. In analogy to bank clients who get into trouble when not paying their invoices promptly, increasing technical debt signifies software decay. However, development teams can act on Technical Debt by removing issues and inefficiencies.

In this paper we presented the results of a case study on fifty-seven Java projects from the Apache Ecosystem, focusing on the amount of TD that is paid back and the issues that are fixed. The results revealed that: a) a large percentage of TD is paid back during the evolution; however, for the majority of the projects the fixing rate is below 30%, b) a small percentage of issue types (such as complex methods, duplicates and exception handling problems) account for the majority of issue fixes and the majority of TD that is paid back and c) although some issue types live in the systems for several years, for the majority of the issues that get fixed, their elimination occurs within one year from their introduction. Such findings can be of help to development and maintenance teams by pinpointing the types of issues that have been adopted by other projects and yielded the largest benefit in terms of Technical Debt reduction.

Finally, although this study investigates a large number of open-source software projects and the issues that are fixed over time, it is still limited to a random sample of projects from only one ecosystem. Furthermore, it relies on SonarQube in order to detect the type of issues that are fixed and the amount of TD that is paid back. It will be valuable, in the future, to analyze a bigger number of projects, from more than one ecosystem, and projects that are developed in other languages or even multilingual projects. Moreover, we plan to replicate this study on all commits of the master branch including build information and performing further data sanitization.

Acknowledgments

Work reported in this paper has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 780572 (project SDK4ED).

References

[1] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *Ieee software*, vol. 29, no. 6, pp. 18–21, 2012.

[2] 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.

[3] M. Lungu, "Reverse engineering software ecosystems," Ph.D. dissertation, University of Lugano, Nov. 2009. [Online]. Available: http://scg.unibe.ch/archive/papers/Lung09b.pdf

[4] K. Manikas, "Revisiting software ecosystems research," *J. Syst. Softw.*, vol. 117, no. C, pp. 84–103, Jul. 2016. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2016.02.003

[5] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "The evolution of project inter-dependencies in a software ecosystem: The case of apache." in *ICSM*, 2013, pp. 280–289.

[6] 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 *Proceedings of the 2009 3rd international symposium on empirical software engineering and measurement*. IEEE Computer Society, 2009, pp. 390–400.

[7] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. IEEE, 2012, pp. 411–416.

[8] 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.

[9] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, "Investigating the impact of design debt on software quality," in *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM, 2011, pp. 17–23.

[10] 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," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 403–414.

[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] B. Curtis, J. Sappidi, and A. Szynkarski, "Estimating the size, cost, and types of technical debt," in *Proceedings of the Third International Workshop on Managing Technical Debt*. IEEE Press, 2012, pp. 49–53.

[13] E. d. S. Maldonado, R. Abdalkareem, E. Shihab, and A. Serebrenik, "An empirical study on the removal of self-admitted technical debt," in *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 2017, pp. 238–248.

[14] G. A. Campbell and P. P. Papapetrou, *SonarQube in Action*, 1st ed. Greenwich, CT, USA: Manning Publications Co., 2013.

[15] A. Decan, T. Mens, M. Claes, and P. Grosjean, "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, Mar. 2016, pp. 493–504.

[16] R. Robbes, M. Lungu, and D. Roethlisberger, “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’12)*, pp. 56:1 – 56:11.

[17] E. Wittern, P. Suter, and S. Rajagopalan, “A look at the dynamics of the javascript package ecosystem,” in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR ’16. New York, NY, USA: ACM, 2016, pp. 351–361. [Online]. Available: http://doi.acm.org/10.1145/2901739.2901743