Characterizing Technical Debt in Evolving Open-source Software

Arthur-Jozsef Molnar*[a] and Simona Motogna[b]
Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania

Keywords: Technical Debt, Software Evolution, Longitudinal Case Study, Open-source Software, Software Maintenance.

Abstract: Technical debt represents deficiencies in software design or implementation often caused by prioritizing feature development over fixing existing issues. Like its financial counterpart, technical debt comprises a principal and an interest. Not addressing it in time leads to development crises, where focus and resources must be shifted to address existing issues. Existing software tools allow measuring the level of debt and pinpointing its sources, which can help practitioners control it. In the present paper we aim to investigate the prevalence, characteristics, and evolution of technical debt in several open-source applications. We used SonarQube to study 112 application versions that covered more than 15 years of development for each application. We studied the way debt characteristics and source code distribution evolved over the target applications' lifecycles. We addressed concerns regarding the accuracy of the analysis and illustrated some of the limitations of existing tools. We observed that a small number of issue types were responsible for most of the debt. We found that each application had its own technical debt particularities. As future work, we aim to expand our selection of analysis tools, leverage open data sets, and extend our investigation to other systems and types of software.

1 INTRODUCTION

The size and complexity of software systems are reflected in their development processes. Stakeholder pressure for continued delivery of features, combined with time and budgetary restrictions can lead to neglecting internal quality attributes such as maintainability, performance or reliability. This prioritization of features over quality was first named by Cunningham in 1992 (Cunningham, 1992), who coined it *technical debt* (TD), in similarity with financial debt. It refers to the typical undesirable outcomes of this prioritization and highlights the difficulties of carrying out long-term software development and maintenance (Fowler, 2019). TD is divided into the principal, which represents the effort needed to recover the original debt, and the interest, which represents the supplementary effort that corresponds to additional system modifications that were caused by the TD principal. When left to accumulate over time, TD can generate crises that halt development (Martini et al., 2015) or lead to severe consequences (US. Securities and Exchange Commission, 2013; Fatemi, 2016).

Authors of (Li et al., 2014) provide a taxonomy that classifies TD according to ten coarse-grained types, which are then further subdivided. Of these, the authors show code TD to be the one most widely mentioned, followed by architectural, testing and design debts. Significant effort towards addressing TD was made over the last decade, with both the research and practitioner communities most often targeting code and architectural debt. Several TD models were proposed such as SQALE (Letouzey, 2012), CAST (Curtis et al., 2012) and SIG (Nugroho et al., 2011). Tools such as SonarQube, NDepend, Kiuwan or CAST Software were developed to provide solutions for estimating and prioritizing it.

Even so, there still exist open issues that need to be addressed. One of them refers to ad-hoc and informal decisions regarding debt management caused by different interpretations from researchers, developers and managers (Klinger et al., 2011). In many cases, this results in TD being ignored for as long as possible, leading to *"the crisis model"* (Martini et al., 2015), where feature development is interspersed with periodic crises during which TD levels must be lowered.

Other open problems regard insufficient evidence in several areas of handling TD such as characterizing its evolution over time, taking into account differ-

[a] https://orcid.org/0000-0002-4113-2953
[b] https://orcid.org/0000-0002-8208-6949
*The publication of this article was supported by the 2021 Development Fund of the Babeş-Bolyai University.

Molnar, A. and Motogna, S.
Characterizing Technical Debt in Evolving Open-source Software.
DOI: 10.5220/0011073600003176
In *Proceedings of the 17th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2022)*, pages 174-185
ISBN: 978-989-758-568-5; ISSN: 2184-4895

ences between programming paradigms, languages, or the specifics of different application types. Providing evidence supported by data sets can significantly increase the quality of TD evaluation as well as stakeholder trust in the available tools. Most case studies targeting TD do not provide a longitudinal view, and very few cover the entire lifespan of their target applications. Likewise, while recent research does include a fine-grained analysis of TD sources (Walkinshaw and Minku, 2018; Lenarduzzi et al., 2020a; Baldassarre et al., 2020; Lenarduzzi et al., 2020b), this is not complemented by an analysis regarding its diffusion across application components or its evolution. Our goal is to provide a long-term view and detailed characterization of source code technical debt over the entire lifetime of the target applications. We aim to focus on how technical debt evolves within the target applications and to compare our results with those from existing research.

Our main contributions are to (i) provide a characterization of technical debt in real-life, evolving open-source software; (ii) investigate how technical debt is diffused across applications and application packages and (iii) study how the composition and distribution of technical debt evolve over the long-term.

2 BACKGROUND

2.1 Technical Debt

The last ten years were marked by a continuous interest in the research pertaining to the prevalence and effects of TD. Several of the proposed methodologies for measuring TD were implemented in the form of software tools. These include the SonarQube[1] platform and NDepend[2], both of which are based on the SQALE method introduced in 2011 (Nugroho et al., 2011). These tools implement the SQALE method by estimating the time required to fix technical debt items associated with source code. The total time required to fix all issues represents the TD principal. Other significant approaches include CAST Software[3], which is based on the eponymous model proposed in 2012 as well as the SIG method[4] and Kiuwan[5], both of which implement their own models for TD estimation. The research community systematically publishes case studies, experience reports and comparative evaluations (Izurieta et al., 2017; Strečanský et al., 2020; Avgeriou et al., 2021; Verdecchia et al., 2018; Zazworka et al., 2013) where these models and corresponding tooling are evaluated.

Most of the available tools that provide quantitative evaluation employ static code analysis to identify existing issues based on a number of predefined rules that correspond to different quality factors such as maintainability, reliability, security or performance. Issues are usually weighted by their severity, their remedial effort is estimated and analyzed code is categorized according to the underlying model employed by the tool. However, the validity and trustworthiness of these models and their corresponding tools remain open research issues due to differences in the identification and estimation of internal quality attributes together with the inherent limitations of static analysis (Strečanský et al., 2020). Recent research has highlighted that in many cases the estimation of remedial effort lacks accuracy and that generated issues do not translate to actual software faults (Lenarduzzi et al., 2020a; Lenarduzzi et al., 2020b). Current key recommendations (Lenarduzzi et al., 2020a; Avgeriou et al., 2021) are that each organization should tailor how it configures and uses these tools according to its own requirements.

2.2 SonarQube

SonarQube is one of the most widely used static analysis platforms for code quality and security, as reflected both in the research literature as well as online (Avgeriou et al., 2021). It integrates the SQALE method and is available in several editions. Its free and open-source *Community Edition* enables analyzing source code in 17 languages, while commercial versions extend this to 29.

Language-based analysis features are implemented in the form of plugins, which facilitates adding support for new languages or improving the accuracy of analysis for those already supported. Each plugin defines a number of language-specific rules, to which source code must conform. When static analysis determines that a rule was violated, an issue is generated. Issues are characterized by their exact location in source code and information they inherit from the generating rule. These include the type, severity, tags and remediation time. Issue type is one of *code smell* (maintainability domain), *bug* (reliability domain), *vulnerability* and *security hotspot* (security domain). For example, Java rule *java:S110*[6] described as *"Inheritance tree of classes should not*

[1]https://www.sonarqube.org/

[2]https://www.ndepend.com/

[3]https://www.castsoftware.com/

[4]https://www.softwareimprovementgroup.com/services/software-risk-assessment/

[5]https://www.kiuwan.com/

[6]Documentation for all rules is available within a deployed SonarQube instance or at https://sonarcloud.io/

be too deep" generates code smells of major severity tagged *"design"*. Remediation time is estimated to be a constant 4 hours to which a 30 minute offset is added for each inheritance level beyond a configurable threshold value.

SonarQube considers total technical debt as the time required to fix all detected code smells. Similar to existing research (Lenarduzzi et al., 2020a; Lenarduzzi et al., 2020b) and in consistency with recent standards (ISO, 2021), we extend the definition to include all detected issues. As such, we refer to technical debt as the time required to fix all the source code issues detected by SonarQube across the maintainability, reliability and security domains. This time is normalized to account for application size in the form of the *technical debt ratio*: $TDR = \frac{TechnicalDebt}{DevTime}$. *DevTime* represents the estimated time required to develop the system from scratch, with 30 minutes required to develop 1 line of production-level code. The technical debt ratio is then assigned a SQALE rating between A (best, $TDR < 5\%$) and E (worst, $TDR \geq 50\%$).

3 RELATED WORK

A recent systematic literature review (Lenarduzzi et al., 2021) classified different contributions to the domain of TD into prioritization or estimation of TD, comparison of different estimation approaches and cross-sectional or longitudinal studies. The study also concluded that most literature results represented scattered approaches to TD prioritization and that there remained a significant lack of empirical evidence in tackling TD.

Other works have carried out a comparative evaluation between different estimation methods such as SQALE, the SIG Method, Quamoco or the Maintainability Index (Izurieta et al., 2017; Strečanský et al., 2020). Authors of (Amanatidis et al., 2020) evaluated the level of agreement between CAST, SonarQube and Squore tooling. Statistical correlation revealed strong agreement when the tools were applied to 50 open source Java and JavaScript programs.

Long term evaluation of three large open source Java projects was reported in (Molnar and Motogna, 2020a). The study compared SonarQube TD estimation with the Maintainability Index as well as a metric based method. Authors concluded that SonarQube was the most accurate of the evaluated methods in estimating overall TD but that it could be surpassed in detecting debt hot spots.

We found that many of the published results were based on cross sectional analyses of TD. In (Alfayez et al., 2018), authors used 91 open source Java projects to investigate the influence of different application characteristics on TD. They concluded that size, application domain, number of releases and commits all had a significant impact on TD, while no significant relation was identified between TD and the number of contributors, branches, or decisions regarding development or management. An investigation performed on 21 open source Java systems (Lenarduzzi et al., 2020a) analyzed the fault proneness of SonarQube rules. The approach showed that only a small number of them were harmful and that most of the issues classified as bugs were not actually fault inducing.

Results from an interview-driven empirical study (Klinger et al., 2011) showed that overall system development was influenced by the introduction of both intended and unintended TD, that TD management was in many cases ad-hoc and carried out without any formalization or a long-term prediction of its impact. A multiple case study (Martini et al., 2015) accompanied by interviews reported that several factors were responsible for architectural TD and that in many cases corrective actions were only applied as a reaction to development problems. Authors also evaluated several strategies for long-term TD management, concluding that periodical efforts to address TD through refactoring could lower its impact over the long term and allow the development process to continue uninterrupted. Existing literature also includes longitudinal approaches that studied TD. In (Besker et al., 2018), authors used surveys and interviews to carry out a study which revealed that on average, 23% of developer time was wasted to TD, with the interest on accumulated debt playing an important role. The paper highlighted the importance of introducing strategies to track and manage both the principal and the interest of debt.

Open source projects represent good candidates for empirical research, as they facilitate direct access to source code and complete information about their development history is usually available. We find most contributions either take a cross-sectional approach (Lenarduzzi et al., 2020a; Baldassarre et al., 2020), or they are short-term longitudinal studies (Nayebi et al., 2019; Lenarduzzi et al., 2020b). Our approach includes all released versions of the studied applications, covering their entire development history. We employ recent work on empirical evaluation of software quality (Molnar and Motogna, 2017; Lenarduzzi et al., 2020a; Molnar and Motogna, 2020b; Baldassarre et al., 2020; Lenarduzzi et al., 2020b) in order to place these results into the proper context, while the publicly accessible open data pack-

age (Molnar, 2022) facilitates replicating or extending our study, as well as carrying out a comparative evaluation against other existing or future studies.

4 CASE STUDY DESIGN

The present work is in line with existing research regarding the prevalence and long-term evolution of source code maintainability (Molnar and Motogna, 2017; Kapllani et al., 2020) and TD (Molnar and Motogna, 2020b; Nayebi et al., 2019; Arif and Rana, 2020). We aimed to improve existing results in the study of open-source software TD. As such, each step of our case study was carried out with consideration to existing results, which we refer to when appropriate.

We used current best practices from case study research (Runeson et al., 2012) in order to design, carry out and report our results. We ensured that our work observed the ACM SIGSOFT's standards for empirical research (Ralph, 2021), as well as the methodological guidelines for longitudinal research described in (Kehr and Kowatsch, 2015). We refer to them when appropriate.

4.1 Objective

The main objective of our case study, defined according to the goal question metric approach (Caldiera and Rombach, 1994) is to *"investigate source code technical debt for the purpose of evaluating its prevalence, characteristics and evolution in the context of open-source software"*.

We operationalize our main objective using the following research questions:

RQ$_1$: *What is the distribution and composition of source code technical debt?* Technical debt is usually introduced due to time and budgetary limitations (Lenarduzzi et al., 2019a); as such, changes to application architecture, development pushes for additional features (Molnar and Motogna, 2017; Martini et al., 2015) or refactoring efforts have a more pronounced effect than software size (Molnar and Motogna, 2020b). Existing research has shown that a small number of files were responsible for the majority of defects in several large systems (Walkinshaw and Minku, 2018). In the case of TD, the top ten most violated SonarQube rules generated 39% of the issues reported in a large-scale case study (Baldassarre et al., 2020), as well as over 50% of the TD in an empirical study (Molnar and Motogna, 2020b) that targeted open-source software.

We aim to go one step further and characterize TD in terms of composition and distribution not just at application, but also at package level. We aim to investigate how debt is distributed across application packages and whether its composition at application level is maintained at finer granularity.

RQ$_2$: *How does technical debt distribution and composition evolve over the long term?* The following step was to investigate the evolution of TD characteristics over the application's entire lifetime. An industry survey (Besker et al., 2018) revealed developers wasted 23% of the time to TD and were forced to introduce new TD due to already existing issues. An investigation of architectural TD (Martini et al., 2015) revealed that its long-term accumulation led to crises where development was halted until debt was reduced to manageable levels.

Through RQ_2 we aimed to improve the long-term characterization of TD by investigating how factors leading to important changes in debt levels affected its distribution and composition. Previous research showed early application versions to be flaky with regards to their maintainability (Molnar and Motogna, 2017) and TD profiles (Molnar and Motogna, 2020a). Do early application versions provide any indication of the TD characteristics of later versions?

For both *RQ*s we carried out a comparative evaluation across the studied applications. We aimed to determine which of our findings might be indicative of larger trends worth further investigation, as well as to determine any particularities of these applications that might improve our understanding of the relation between TD and software evolution.

4.2 Data Collection

The present work is a finer grained investigation based on previous results (Molnar and Motogna, 2020b). We maintain the selection of target applications, which we elaborate in this section. Our literature survey identified an abundance of empirical research targeting complex, widely-used open-source or commercial systems. Yet, we found research considering software evolution, especially over the long-term to be under-represented. This was especially the case for GUI-based applications. As such, we decided to include GUI-driven applications having a well-documented development period and a consistent user base. We restricted the selection to Java software, as it enabled comparative evaluation against a broad range of existing results. We excluded applications with dependencies to external software or hardware. We disregarded applications that went through lengthy development hiatuses, or which were aban-

Table 1: Details for the earliest and most recent version for each application in our study.

Application	Version (Released)	Packages	LOC	Issues (SQALE rating)			TD
				Bugs	Vulnerabilities	Code Smells	work days
FreeMind	0.0.3 (July 2000)	5	2,770	3 E	0 A	248 A	3
	1.1.0Beta2 (Feb 2016)	31	43,269	88	2 E	4,521	78
jEdit	2.3pre2 (Jan 2000)	10	22,311	62 E	0 A	1,427 A	30
	5.6.0 (Sep 2020)	31	96,912	270	4 E	8,807	130
TuxGuitar	0.1pre (June 2006)	30	8,960	34 E	2 E	1,188 A	17
	1.5.4 (May 2020)	239	106,457	170	13	3,692	73

doned by their original developers (Klinger et al., 2011).

Our process resulted in the inclusion of the *FreeMind*[7] mind-mapper, the *jEdit*[8] text editor and the *TuxGuitar*[9] tablature editor.

FreeMind is a popular mind-mapping application with a rich user base and a plugin environment. Its first version comprised 5 packages and 2,770 lines of code[10]. This made it the least complex release in our study, which was visible at functionality and user experience levels. Future versions have improved upon it greatly, with versions 0.8.0 and 1.0 updating the user interface and introducing many new functionalities. We found a $2\frac{1}{2}$ year development hiatus after version 0.8.0, after which development continued until the most current version, 1.1.0Beta2. While several forks exist[11], FreeMind itself has remained popular, having an active user forum and recording 530k downloads over the last year and over 25 million during its lifetime[12].

jEdit is a mature text editor targeted towards software developers. Its first publicly available version was 2.3pre2, released in January 2000. Comprising over 22k lines of code organized across 10 packages, it is the most mature initial version in our study. As the application did not record development hiatuses or large-scale refactoring, its development appears steady and consistent across releases, with the most recent version released in September, 2020. jEdit provides a plugin environment and enjoys continued popularity attested by over 86k downloads within the last year, and 9.1 million over the application's lifetime.

TuxGuitar is a multi-track tablature editor that supports several GUI toolkits. We tested its releases using the Standard Widget Toolkit. As illustrated in Figure 1, TuxGuitar and jEdit had a similar evolution marked by incremental development. One difference is the development hiatus between versions 1.2 and 1.3, after which development continued, with the latest version released in mid-2020. In the case of TuxGuitar we counted 209k downloads within the last year and 7.1 million over its entire lifetime. Given the number of plugins included in the default download, TuxGuitar has the largest number of packages, as well as the most important package-level changes of the studied applications. Its initial version comprised 30 packages and 8,960 lines of code, with over 350 packages used within the application's lifetime.

In addition to being widely used, several releases of FreeMind and jEdit were used in previous research targeting GUI testing (Arlt et al., 2012; Yuan and Memon, 2010) and software quality (Molnar and Motogna, 2017; Kapllani et al., 2020).

Existing research used either commit (Walkinshaw and Minku, 2018; Lenarduzzi et al., 2020a; Lenarduzzi et al., 2019b; Lenarduzzi et al., 2020b) or release (Izurieta et al., 2017; Baldassarre et al., 2020) level granularity for collecting target application data. Given the timespan and large number of application releases, we included all publicly-released versions. This provided sufficient measurement waves (Kehr and Kowatsch, 2015) to provide answers to the proposed RQs, while helping to mitigate issues such as missing dependencies, misconfiguration or even compilation errors, which can sometimes be found in open-source software (Barkmann et al., 2009).

In the case of both FreeMind and jEdit, we found several preview versions that were released in the span of a few days; in this case, we decided to include the last release from each series, in order to keep the number of included versions manageable. This resulted in a total of 112 releases, comprised of 38 FreeMind, 46 jEdit and 28 TuxGuitar versions.

We manually examined and compiled each version's source code. We found TuxGuitar versions were distributed together with a number of plugins, which we considered part of the application and included in our study. We also discovered instances where library source code was packaged with application code, such as the *com.microstar.xml* parser or the *BeanShell* interpreter present in some jEdit ver-

[7]http://freemind.sourceforge.net/

[8]http://jedit.org

[9]http://www.tuxguitar.com.ar

[10]All metrics were recorded using SonarQube 9.0.1

[11]https://www.freeplane.org/

[12]Download data points recorded on September 29, 2021 through SourceForge

sions. We separated this code into external libraries that were added to the classpath. Existing research showed that up to 32% (Barkmann et al., 2009) of open-source software required manual fixes to compile and run; we thoroughly tested that each version in our study compiled correctly and could be executed with full functionalities available.

Figure 1 illustrates application versions in our study and their release dates. We refer to application releases using the version numbers their developers assigned to them; at the time of writing, all application releases remain available for download and can be identified by their version. However, in our analysis we do not attribute additional significance to version numbers and treat each release in the same way. We also observed the existence of several development hiatuses between some of the studied versions, as discussed in the previous sections. Again, we did not factor this into our analysis.

4.3 Data Analysis

We analyzed each application version using a local instance of SonarQube 9.0.1 that we configured for historical analysis. This enabled tracking the location and status of each detected issue across application versions; for each version, we determined the proportion of newly introduced debt versus debt that was carried over from the previous version. The vertical bar corresponding to each application release in Figure 1 indicates its TDR, with the darker segment illustrating the portion of debt ratio newly added in that version. All TD was considered to be newly added in the first release of an application. For context, Table 1 provides more detailed information regarding the earliest and latest version for each application included in our study.

We employed the default SonarWay profile, ruleset and associated analysis thresholds. We restricted our investigation to the Java language. This resulted in a total of 55,630 issues, estimated at 1,081 days of TD. These were broken down into 3,131 bugs, 30 vulnerabilities and 52,469 code smells. During the monitored development period, 38,153 of these issues were fixed or they no longer appeared in more recent application versions.

As SonarQube automatically purges detailed issue and metric information for older analyses, we set up a second instance, which was configured to treat each version as a stand-alone application. We used this second instance to record and persist detailed file and package-level metrics across all application versions (SonarSource, 2021).

Data analysis was carried out using a number of custom-written Python scripts that employ the SonarQube API to extract and process information on source files, rules and detected issues. We prepared an open data package (Molnar, 2022) that allows replicating or extending our investigation. It includes the database files for the required instances of SonarQube, the analysis scripts used together with their output and instructions for configuring the environment and running the analyses.

5 RESULTS AND DISCUSSION

In this section we detail and discuss our results, organized according to the previously defined research questions.

RQ_1: What is the Distribution and Composition of Source Code Technical Debt?

Figure 1 illustrates that most application versions have a $TDR < 5\%$ and receive an A rating according to SQALE. The only exception were FreeMind versions 0.8.*, where significant additional debt was accrued. The amount of technical debt present also increased significantly in jEdit version 4.0pre4, where important new functionalities were introduced. In the case of TuxGuitar, early versions already presented a $TDR < 5\%$, and remained that way in all subsequent versions.

Next, we investigated how TD was diffused across application packages. We use Figure 2 to illustrate how TD was distributed across the most debt-intensive packages for each application; the full data set remains available in our open data package (Molnar, 2022). In the case of FreeMind and jEdit, over 80% of the total debt was diffused across nine and eight packages, respectively. In both applications, debt distribution followed the Pareto principle, with a few code-heavy packages also carrying most of the existing debt. In contrast, each TuxGuitar plugin had its own package structure. This contributed to the large number of application packages illustrated in Figure 2. It also meant that both source code as well as debt were distributed more evenly across application packages. The 21 TuxGuitar packages illustrated in Figure 2 only carried half of the application's total debt. This is in line with previous findings (Molnar and Motogna, 2020b) that showed tight correlation between the amount of TD and lines of code count at file level.

Next we drilled down to SonarQube rule level.

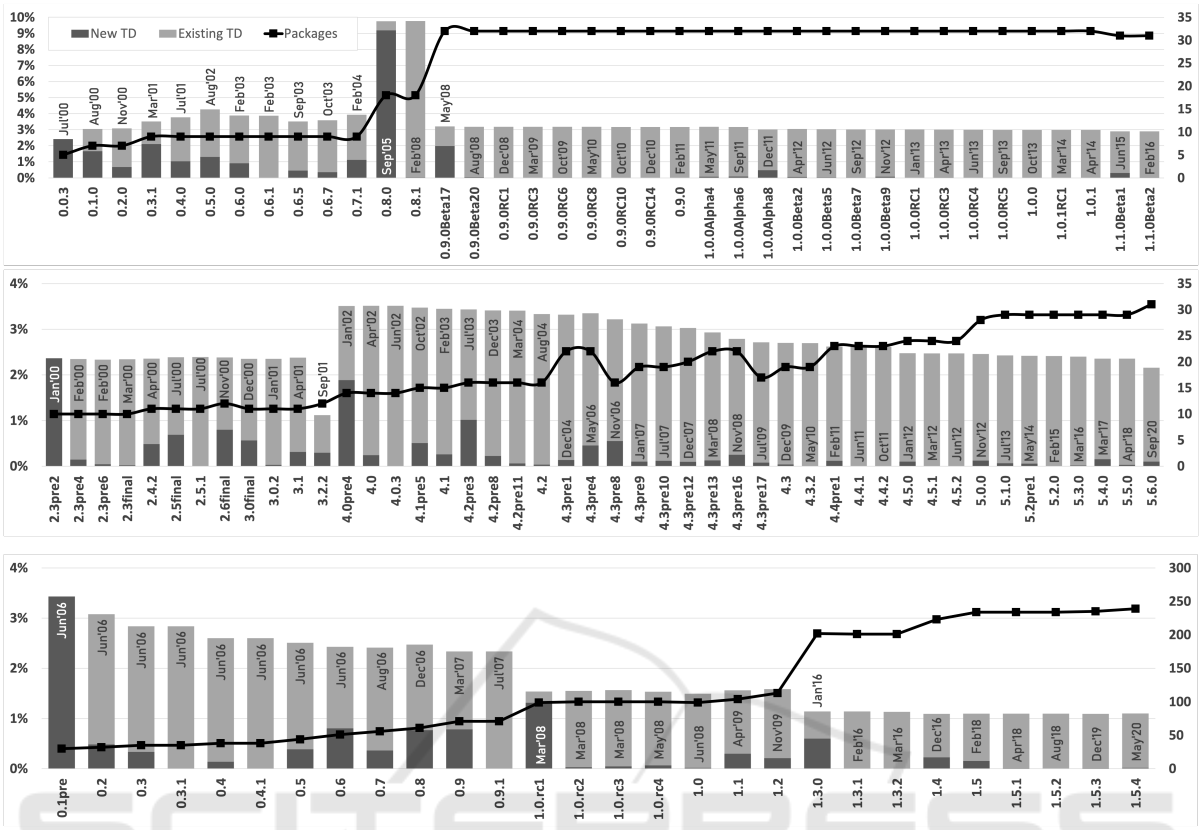

Figure 1: Studied versions of FreeMind (top), jEdit (middle) and TuxGuitar (bottom); each bar represents the technical debt ratio for that version, with the darker segment showing the debt newly added to that version (left-side scale). The dotted line represents the number of packages (right-side scale).

We confirmed that a small number of rules together generated most of the estimated technical debt. For each application, we considered the top 10 such rules which we illustrated in Table 2. We note the significant degree of overlap, as the selection of 15 rules illustrated in Table 2 generated more than two thirds of the TD in each application.

We also noted that the rules which generated most of the TD were application specific, and we could not identify a single rule that was dominant across all three applications. For example, rule *java:S1948* generated the most TD in FreeMind after code duplication. Its root cause was the presence of non-serializable members in serializable classes; this is a potential bug and security vulnerability in case containing instances are serialized. Another prevalent issue was code duplication, which generated significant debt in FreeMind and TuxGuitar. In the case of FreeMind, this was the dominating source of debt in versions 0.8.*, where it was concentrated in the *freemind.controller.actions.generated.instance.impl* package. The spike shown in Figure 2 was caused by a large amount of generated code that included duplicated sections, but also non-standard variable names, usage of labels (*java:S1119*), classes specific to a vendor's implementation of Java (*sun.** packages, *java:S1191*), loops with more than one *break* or *continue* statements (*java:S135*) and other issues.

We compared the composition of debt in the studied applications against that uncovered in recent existing research. In (Lenarduzzi et al., 2020b), a case study targeting 33 Java projects from the Apache Foundation analyzed the change proneness of classes encumbered by TD. Authors found *duplicated* code to be the most often encountered code smell. Three of the 10 most commonly violated rules in (Lenarduzzi et al., 2020b) (*duplicated*, *java:S1192*, *java:S1948*) were also well represented in our results.

In (Baldassarre et al., 2020), authors also provided details regarding the type and severity distribution of SonarQube issues, which we found similar to previous results (Molnar and Motogna, 2020b). Authors also provided a list of the 22 rules that each generated more than 300 issues. We found the following eight also present in our results in Table 2: *java:S125*, *duplicated*, *java:S3776*, *java:S1948*,

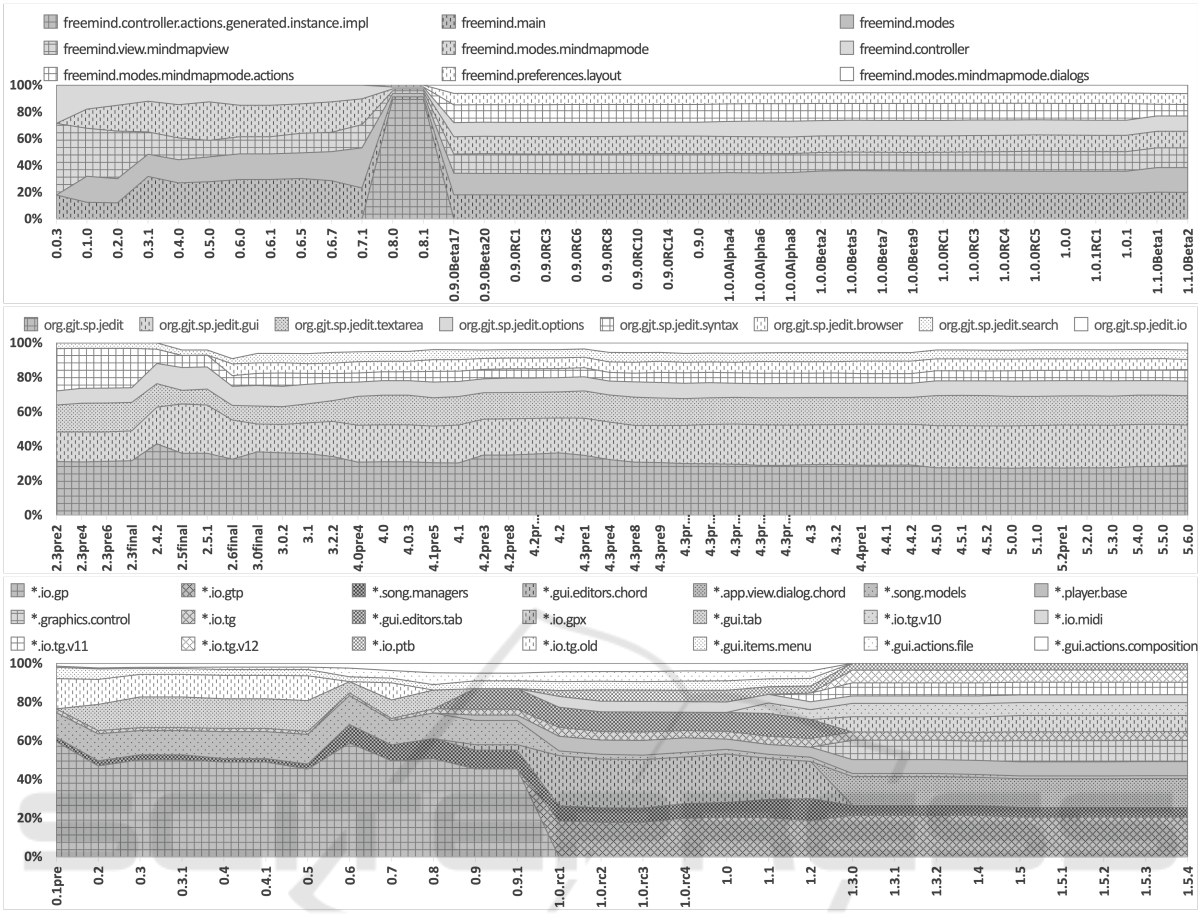

Figure 2: Per package TD in FreeMind (top), jEdit (middle) and TuxGuitar (bottom, * stands for *org.herac.tuxguitar*) representing 80% of total TD for FreeMind and jEdit, and 50% for TuxGuitar; vertical scale relative to represented data for legibility.

java:S1104, *java:S1192*, *java:S2696* and *java:S1117*.

In (Lenarduzzi et al., 2020a), authors used machine learning to investigate whether SonarQube issues predicted actual software faults. While a reliable relation between most issues and documented faults could not be discovered, a number of rules were found to have some predictive power. Among them, *java:S125*, *java:S1192* and *java:S1117* are also present in Table 2.

Most of the Apache systems studied in these works were not GUI-driven, which could represent an important difference regarding the type and distribution of TD. There were also differences in how TD was measured across these studies; our approach was to measure TD in minutes, while these studies defaulted to counting the number of issues generating it. While these differences did not allow us to draw definitive conclusions, we observed the existence of overlap in the composition of TD across the studied systems.

Empirical research has shown that a limited selection of rules remained responsible for most of the debt accrued in the studied applications. However, this selection remains application-specific. As such, when making decisions based on the output of analysis tools, developers should be aware both of the target application's profile as well as the fault-predictive power of the rules responsible for the reported debt.

RQ_2: *How Does Technical Debt Distribution and Composition Evolve over the Long Term?*

We started by investigating the relation between lines of code and the amount of TD present. For each application version, we calculated the Spearman rank correlation between lines of code and TD effort at package level. We found a high correlation, with mean values of $\rho > 0.8$ and $\sigma < 0.067$ considered across each application's versions. The common trend was that of lower, but still significant ($\rho > 0.6$) correla-

Table 2: Percentage of total TD generated by the top 10 rules in each application. Critical issues in bold.

Rule ID	FreeMind	jEdit	TuxGuitar	Description
java:S125	4.35%	34.86%	1.75%	Sections of code should not be commented out
duplicated	11.71%	2.23%	23.47%	Source files should not have any duplicated blocks
java:S3740	10.22%	3.59%	8.71%	Raw types should not be used
java:S3776	**3.06%**	**6.24%**	**9.72%**	**Cognitive Complexity of methods should not be too high**
java:S1948	**11.28%**	**5.53%**	**0.42%**	**Serializable class fields should either be transient or serializable**
java:S1149	7.47%	5.57%	0.63%	Synchronized classes (e.g., Vector, Hashtable) should not be used
java:S1874	4.43%	5.47%	1.85%	"@Deprecated" code should not be used
java:S1161	4.48%	2.94%	4.03%	"@Override" annotation should be used
java:S1854	1.46%	0.76%	5.20%	Unused assignments should be removed
java:S1604	0.75%	0.80%	4.54%	Single-method anonymous inner classes should become lambdas
java:S1104	3.10%	2.29%	0.52%	Class variable fields should not have public accessibility
java:S1905	0.63%	0.29%	3.80%	Redundant casts should not be used
java:S1192	**1.34%**	**2.30%**	**0.51%**	**String literals should not be duplicated**
java:S2696	**2.66%**	**0.83%**	**0.22%**	**Instance methods should not write to "static" fields**
java:S1117	0.32%	0.65%	2.09%	Local variables should not shadow class fields
TOTAL	**67.27%**	**74.32%**	**67.47%**	

tion in early application versions, followed by high correlation in mature versions. This confirmed previous observations (Molnar and Motogna, 2020b) regarding the increased volatility of software quality in early application versions. It also followed existing results (Walkinshaw and Minku, 2018) that showed packages rich in source code accumulated the majority of observed issues.

We reported on the composition of TD across each application's version in our answer to RQ_1. We then carried out a package-level analysis in order to determine whether the distribution and composition of TD remained consistent across application versions at the package level. If that was not the case, we aimed to determine the reason for the change and its magnitude. The first step was to calculate the contribution of each package to the total TD. For each pair of consecutive releases (e.g. FreeMind 0.8.1 and 0.9.0Beta17), we correlated TD levels across packages using Spearman's rank correlation. Spearman correlation highlights changes in rank, namely situations where one or more packages accumulate or shed TD faster than others. We obtained very high correlation ($\rho > 0.9$) across most version pairs, except the following versions and their immediate predecessors: FreeMind 0.3.1 ($\rho \approx 0.4$), jEdit 2.4.2 ($\rho \approx 0.8$) and TuxGuitar versions 1.0.rc1 ($\rho \approx 0.45$). We ascribe these changes to the introduction of new TD as shown in Figure 1 and discussed in our answer for RQ_1. Remaining version pairs, including all jEdit releases showed TD to be very consistently distributed across source code packages.

Figure 1 shows that the versions highlighted above accrued significant additional debt. While additional debt was taken on in FreeMind 0.9.0Beta17, TuxGuitar 1.0.rc1 and 1.3.0, refactoring helped improve the overall *TDR* of these versions. In the case of FreeMind, the large swath of TD incurred in version 0.8.0 was resolved in 0.9.0Beta17 with the elimination of the offending package and code refactoring. The introduction of additional debt in the later version was due to development work resulting in additional functionalities.

In the case of jEdit, we earmarked version 4.0pre4 for further examination, due to its increased *TDR*. We found it to be an important update that included additional functionalities such as improved text area management, buffer events, an improved document model and user interface updates. We examined the debt composition of this version and discovered that its increased debt ratio was due to a plethora of new issues, many of which were generated by the inclusion of commented code (*java:S125*). Closer examination uncovered that most of these comments were used for delimiting sections of the source code such as methods or conditional statement branches using an annotation specific to jEdit itself, and were erroneously registered as deprecated code by the static analyzer.

As the previous test considered total TD for each package, we also carried out a fine grained examination where we examined whether debt composition varied across releases in individual source code packages. As expected, we discovered the composition of TD present in most packages to be consistent across versions. For each application we encountered versions where packages were eliminated, introduced or refactored. One example of the latter was TuxGuitar package *org.herac.tuxguitar.io.gp*, which carried an important share of TD up to version 1.0.rc1, where it was refactored into *org.herac.tuxguitar.io.gtp*; interestingly, as illustrated in Figure 2 the refactored package carried less than half of the initial version's debt, even without significant changes to the number of lines of code.

Our conclusion for RQ_2 is that the distribution and

composition of TD remained generally stable across multiple application versions. Notable exceptions were early application versions that showed increased volatility and versions where major refactoring or development work have taken place.

6 THREATS TO VALIDITY

Our study was conducted based on existing best practices for empirical (Ralph, 2021) case study research (Runeson et al., 2012; Kehr and Kowatsch, 2015). We first defined the main objective, established the research questions, carried out the target application selection process, collected, processed and then analyzed the data. We made the collected data from the studied applications public as part of an open data package (Molnar, 2022).

Internal threats were addressed by manually examining and publishing the source code and scripts that were used in data collection and processing. We included sanity checks combined with manual examination of the source code and analysis results. The remaining threat regards our reliance on SonarQube itself. Previous research has shown that differences exist between the results obtained when using different tools (Izurieta et al., 2017; Strečanský et al., 2020). In our experience, using different versions of the same tool can also result in significant differences. In previous work we used SonarQube versions 7.9 (Molnar and Motogna, 2020a) and 8.2 (Molnar and Motogna, 2020b), which target some of the application versions included in our study. We found the presence of additional rules in newer versions, as well as changes to estimated remediation times lead to some changes in reported results. This was most apparent in the case of discovered vulnerabilities, as improved taint analysis introduced in SonarQube 8.5[13] lead to the elimination of many falsely reported issues.

External threats are related to the selection of applications and analysis tools in our study coupled with the inherent limitations of generalizing our conclusions. Restricting our inclusion criteria to a single application type improved data triangulation (Runeson et al., 2012) and the possibility of cross-application comparison. Conversely, it limited extrapolating our results to other system and system types. We addressed external threats by including all released application versions in our study and by validating more general conclusions with relevant results from the literature.

Previous empirical investigations addressed this issue by creating and analyzing comprehensive stores of collected data (Lenarduzzi et al., 2019b; Lenarduzzi et al., 2020b). In this regard, our focus was on longitudinal analysis carried out on a curated set of versions, where the additional requirement of manual code examination made automating the process difficult.

Construct threats were addressed by using SonarQube, a well-known and widely used tool for assessing software quality and security. We employed the default *SonarWay* Java profile. We drilled down to rule level and investigated the significance of our findings. We put our results into the context of similar research (Walkinshaw and Minku, 2018; Lenarduzzi et al., 2020a; Lenarduzzi et al., 2020b).

However, an extended analysis regarding the link between our findings and actual maintainability or fault-proneness was beyond our scope. We believe that SonarQube can provide valuable assistance during the development process, but that the set of rules and their configuration should be customized according to requirements.

The decision to restrict the study to released versions lead to changes discarded between two releases to not be represented in our study. While denser measurement waves (Kehr and Kowatsch, 2015) would have alleviated this risk, we considered that covering all released versions was sufficient to characterize the evolution of debt. Finally, we must mention the existence of the survival bias, represented by the possibility that these applications' TD characteristics and their long lifecycles were not a coincidence.

7 CONCLUSIONS

We carried out a longitudinal case study targeting all the released versions of three complex, open-source applications. In contrast to most existing work (Walkinshaw and Minku, 2018; Lenarduzzi et al., 2020a; Baldassarre et al., 2020; Lenarduzzi et al., 2020b), we investigated the relation between long-term software evolution and TD characteristics. Our study covered at least 15 years of development for each studied application. We carried out a fine grained analysis regarding the diffusion and composition of TD across application source code, and studied the effects of major software changes and refactoring.

We showed that a small number of rules were responsible for generating most of the technical debt found in each application. Comparison with existing results targeting other systems revealed the existence of a subset of rule violations to be the cause of significant debt across many applications. The most

[13]https://www.sonarqube.org/sonarqube-8-5/

prevalent of them were code duplication, high cyclomatic complexity of functions and incorrect use of modifiers in serializable classes. Our examination showed that technical debt fluctuated mostly within early application versions, as they were most affected by both software changes and refactoring. Once core architecture was established, we found very little variance between versions. One possible explanation is that the combination between suitable application architecture and experienced contributors precludes the need for making large-scale changes.

Data collection was carried out using the default rule set and analysis parameters for Java. We believe some of these rules can provide valuable insight into software issues such as increased code complexity (*java:S3776*, *java:S1854* and *java:S1604*), code duplication (*duplicated*, *java:S1192*) or the use of language features in a discouraged way (*java:S1117*). We also discovered problematic issues, such as structured comments that were erroneously classified as commented out code, or custom GUI components flagged as being too deep within their inheritance hierarchy. We mirror the conclusion from (Lenarduzzi et al., 2020a), that organizations should use the platform's customization features in order to select and configure a subset of rules that cover their requirements. Our results can be beneficial to practitioners relying on SonarQube regarding the prioritisation and management of TD in the context of large software systems.

We aim to extend our investigation by providing long-term analyses for other system types. Existing efforts such as the *Technical Debt Dataset* (Lenarduzzi et al., 2019b) or replication packages available for related works (Herbold et al., 2020; Ferenc et al., 2018) can be leveraged for analyses at both release and commit levels.

REFERENCES

Alfayez, R., Chen, C., Behnamghader, P., Srisopha, K., and Boehm, B. (2018). An empirical study of technical debt in open-source software systems. In Madni, A. M., Boehm, B., Ghanem, R. G., Erwin, D., and Wheaton, M. J., editors, *Disciplinary Convergence in Systems Engineering Research*, pages 113–125. Springer International Publishing.

Amanatidis, T., Mittas, N., Moschou, A., Chatzigeorgiou, A., Ampatzoglou, A., and Angelis, L. (2020). Evaluating the agreement among technical debt measurement tools: building an empirical benchmark of technical debt liabilities. *Empir. Softw. Eng.*, 25(5):4161–4204.

Arif, A. and Rana, Z. A. (2020). Refactoring of code to remove technical debt and reduce maintenance effort. In *2020 14th International Conference on Open Source Systems and Technologies (ICOSST)*, pages 1–7.

Arlt, S., Banerjee, I., Bertolini, C., Memon, A. M., and Schaf, M. (2012). Grey-box gui testing: Efficient generation of event sequences. *CoRR*, abs/1205.4928.

Avgeriou, P. C., Taibi, D., Ampatzoglou, A., Arcelli Fontana, F., Besker, T., Chatzigeorgiou, A., Lenarduzzi, V., Martini, A., Moschou, A., Pigazzini, I., Saarimaki, N., Sas, D. D., de Toledo, S. S., and Tsintzira, A. A. (2021). An overview and comparison of technical debt measurement tools. *IEEE Software*, 38(3):61–71.

Baldassarre, M. T., Lenarduzzi, V., Romano, S., and Saarimäki, N. (2020). On the diffuseness of technical debt items and accuracy of remediation time when using sonarqube. *Information and Software Technology*, 128:106377.

Barkmann, H., Lincke, R., and Löwe, W. (2009). Quantitative evaluation of software quality metrics in open-source projects. In *2009 International Conference on Advanced Information Networking and Applications Workshops*, pages 1067–1072.

Besker, T., Martini, A., and Bosch, J. (2018). Technical debt cripples software developer productivity: A longitudinal study on developers' daily software development work. In *Proceedings of the 2018 International Conference on Technical Debt*, TechDebt '18, page 105–114, New York, NY, USA. Association for Computing Machinery.

Caldiera, V. R. B. G. and Rombach, H. D. (1994). The Goal Question Metric approach. *Encyclopedia of software engineering*, pages 528–532.

Cunningham, W. (1992). The wycash portfolio management system. *SIGPLAN OOPS Mess.*, 4(2):29–30.

Curtis, B., Sappidi, J., and Szynkarski, A. (2012). Estimating the size, cost, and types of technical debt. In *2012 Third International Workshop on Managing Technical Debt (MTD)*, pages 49–53.

Fatemi, F. (2016). Technical debt: The silent company killer. https://www.forbes.com/sites/falonfatemi/2016/05/30/technical-debt-the-silent-company-killer/?sh=1578a2ae4562.

Ferenc, R., Tóth, Z., Ladányi, G., Siket, I., and Gyimóthy, T. (2018). A public unified bug dataset for java. In *Proc. of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE'18, page 12–21. ACM.

Fowler, M. (2019). Technical debt. https://martinfowler.com/bliki/TechnicalDebt.html.

Herbold, S., Trautsch, A., Trautsch, F., and Ledel, B. (2020). Issues with szz: An empirical assessment of the state of practice of defect prediction data collection.

ISO (2021). ISO/IEC 5055:2021 standard for automated source code quality measures. https://www.iso.org/standard/80623.html.

Izurieta, C., Griffith, I., and Huvaere, C. (2017). An industry perspective to comparing the sqale and quamoco software quality models. In *2017 ACM/IEEE Intern.*

Symp. on Empirical Software Engineering and Measurement (ESEM), pages 287–296.

Kapllani, G., Khomyakov, I., Mirgalimova, R., and Sillitti, A. (2020). An empirical analysis of the maintainability evolution of open source systems. In *IFIP International Conference on Open Source Systems*, pages 78–86. Springer.

Kehr, F. and Kowatsch, T. (2015). Quantitative longitudinal research: A review of is literature, and a set of methodological guidelines. ECIS 2015 Completed Research Papers.

Klinger, T., Tarr, P., Wagstrom, P., and Williams, C. (2011). An enterprise perspective on technical debt. In *Proceedings of the 2nd Workshop on Managing Technical Debt*, MTD '11, page 35–38. ACM.

Lenarduzzi, V., Besker, T., Taibi, D., Martini, A., and Arcelli Fontana, F. (2021). A systematic literature review on technical debt prioritization: Strategies, processes, factors, and tools. *Journal of Systems and Software*, 171:110827.

Lenarduzzi, V., Lomio, F., Huttunen, H., and Taibi, D. (2020a). Are sonarqube rules inducing bugs? In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 501–511.

Lenarduzzi, V., Orava, T., Saarimaki, N., Systa, K., and Taibi, D. (2019a). An empirical study on technical debt in a finnish sme. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–6, Los Alamitos, CA, USA. IEEE Computer Society.

Lenarduzzi, V., Saarimäki, N., and Taibi, D. (2019b). The technical debt dataset. In *15th Conference on Predictive Models and Data Analytics in Software Engineering*.

Lenarduzzi, V., Saarimäki, N., and Taibi, D. (2020b). Some sonarqube issues have a significant but small effect on faults and changes. a large-scale empirical study. *Journal of Systems and Software*, 170:110750.

Letouzey, J.-L. (2012). The sqale method for evaluating technical debt. In *Proceedings of the Third International Workshop on Managing Technical Debt*, MTD '12, pages 31–36. IEEE Press.

Li, Z., Avgeriou, P., and Liang, P. (2014). A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, pages 193–220.

Martini, A., Bosch, J., and Chaudron, M. (2015). Investigating architectural technical debt accumulation and refactoring over time. *Inf. Softw. Technol.*, 67(C):237–253.

Molnar, A. and Motogna, S. (2017). Discovering maintainability changes in large software systems. In *Proc. of 27th Intern. Workshop on Software Measurement & 12th Intern. Conf. on Soft. Process and Product Measurement*, IWSM Mensura '17, pages 88–93. ACM.

Molnar, A. and Motogna, S. (2020a). Longitudinal evaluation of open-source software maintainability. In *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, pages 120–131. INSTICC, SciTePress.

Molnar, A.-J. (2022). Open Data Package for article "Characterizing Technical Debt in Evolving Open-Source Software". https://doi.org/10.6084/m9.figshare.14601411.v1.

Molnar, A.-J. and Motogna, S. (2020b). Long-term evaluation of technical debt in open-source software. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ESEM '20, New York, NY, USA. Association for Computing Machinery.

Nayebi, M., Cai, Y., Kazman, R., Ruhe, G., Feng, Q., Carlson, C., and Chew, F. (2019). A longitudinal study of identifying and paying down architecture debt. In *Proc. of the 41st Intern. Conf. on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '19, page 171–180. IEEE Press.

Nugroho, A., Visser, J., and Kuipers, T. (2011). An empirical model of technical debt and interest. In *Proceedings of the 2nd Workshop on Managing Technical Debt (MTD '11)*, pages 1–8.

Ralph, P. (2021). ACM SIGSOFT Empirical Standards Released. *SIGSOFT Softw. Eng. Notes*, 46(1):19.

Runeson, P., Host, M., Rainer, A., and Regnell, B. (2012). *Case Study Research in Software Engineering: Guidelines and Examples*. Wiley Publishing, 1st edition.

SonarSource (2021). Managing project history. https://docs.sonarqube.org/latest/project-administration/managing-project-history/.

Strečanský, P., Chren, S., and Rossi, B. (2020). Comparing maintainability index, sig method, and sqale for technical debt identification. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, SAC '20, page 121–124, New York, NY, USA. Association for Computing Machinery.

US. Securities and Exchange Commission (2013). In the matter of knight capital americas llc. https://www.sec.gov/whistleblower/award-claim/award-claim-2013-93.

Verdecchia, R., Malavolta, I., and Lago, P. (2018). Architectural technical debt identification: The research landscape. In *Proceedings of the 2018 International Conference on Technical Debt*, TechDebt '18, page 11–20, New York, NY, USA. Association for Computing Machinery.

Walkinshaw, N. and Minku, L. (2018). Are 20% of files responsible for 80% of defects? In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '18, pages 1–10, New York, NY, USA. Association for Computing Machinery.

Yuan, X. and Memon, A. M. (2010). Generating event sequence-based test cases using GUI run-time state feedback. *IEEE Transactions on Software Engineering*, 36(1):81–95.

Zazworka, N., Vetro, A., Izurieta, C., Wong, S., Cai, Y., Seaman, C., and Shull, F. (2013). Comparing four approaches for technical debt identification. *SOFTWARE QUALITY JOURNAL*, 22:1–24.