


Review

# On the Relationship between Self-Admitted Technical Debt Removals and Technical Debt Measures

Lerina Aversano \* , Martina Iammarino, Mimmo Carapella, Andrea Del Vecchio and Laura Nardi

Department of Engineering, University of Sannio, 82100 Benevento, Italy; iammarino@unisannio.it (M.I.); mimmo.carapella@studenti.unisannio.it (M.C.); andrea.delvecchio@studenti.unisannio.it (A.D.V.); laura.nardi@studenti.unisannio.it (L.N.)

\* Correspondence: aversano@unisannio.it

Received: 30 April 2020; Accepted: 9 July 2020; Published: 11 July 2020



**Abstract:** The technical debt (TD) in a software project refers to the adoption of an inadequate solution from its design to the source code. When developers admit the presence of technical debt in the source code, through comments or commit messages, it is called self-admitted technical debt (SATD). This aspect of TD has been the subject of numerous research studies, which have investigated its distribution, the impact on software quality, and removal. Therefore, this work focuses on the relationship between SATD and TD values. In particular, the study aims to compare the admitted technical debt with respect to its objective measure. In fact, the trends of TD values during SATD removals have been studied. This was done thanks to the use of an SATD dataset and their related removals in four open source projects. Instead, the SonarQube tool was used to measure TD values. Thanks to this work, it turned out that SATD removals in a few cases correspond to an effective reduction of TD values, while in numerous cases, the classes indicated are removed.

**Keywords:** software quality; technical debt; self-admitted technical debt; software maintenance; software evolution; software measures

## 1. Introduction

During the evolution of a software system, several problems can occur leading to the decreasing of the quality measures and introducing technical debt. Some of these approaches are more subjective, while some others are more objective and based on the development assertions. In the literature, there are some approaches based on the assessment of metrics to identify problems concerned with well-known principles of object-oriented design [1,2]. While other studies focused on the architectural-level proposed approach for the identification of technical debt at that level.

In [3], the metrics on a package are used for the identification of architectural technical debt, while in other studies, the comments extracted from the code were used [4,5]. In [6], the authors analyzed the structure of code with the aim of visualizing the technical debt at the architectural level. Differently from these studies in [7], the authors used a scenario of changes to identify technical debt.

Self-admitted technical debt (SATD) refers to the admission of technical debt by developers through the use of comments or references in commit messages. In the literature, several studies have deepened the SATD, evaluating it from different perspectives, such as its diffusion, the consequences that derive from it on the quality of the software, and the consequent removal.

In this document, we examine the removal of SATD to understand if it affects objective measures of technical debt values. In particular, the goal is to verify whether the value of the technical debt for a given file is reduced in the comments in which the developers report their removal.

Therefore, the trends of the technical debt values are evaluated in open source software projects. In particular, the study aims to investigate the differences between the value of the technical debt

obtained through an objective measure, namely measured with the SonarQube platform, and the self-admitted technical debt reported by the developers and collected in a dataset [8].

This dataset labels four different types of change that can be used to remove the technical debt in the source code: class removal, method removal, modified method, and unchanged method.

To perform a deeper investigation on what happens when SATD removals occur, a further analysis has been performed to assess the correlation between the variations of the technical debt and the values of the object-oriented metrics, including Chidamber and Kemerer metrics [9]. The analysis aims to verify if it is possible to observe a consistency change between the results obtained for the technical debt measure changes and the changes in the object-oriented metrics mentioned above.

Section 3 contains the research questions that have been studied. The results obtained show that in the classes in which the removals of SATD were recorded, the value of the technical debt decreases considerably, until it disappears. Moreover, the values of Object-Oriented (OO) quality metrics, including Chidamber & Kemerer metrics, improve their values by a low percentage with respect to the SATD removals considered. These findings can be used for a better understanding of the numerous approaches used for the identification and measurement of technical debt in the software system.

The rest of the paper is structured as follows: Section 2 describes primarily related works, Section 3 defines the design of the study, the obtained results are reported in Section 4, while conclusions and future work are outlined in the last section.

## 2. Related Work

This section reports the literature related to (1) the detection of TD focusing more on “self-admitted” TD; and (2) investigation of effects on SATD and quality metrics.

### 2.1. Detection of SATD

Potdar and Shihab [10] conducted a qualitative analysis on the technical debt (TD) in the source code of open-source projects and observed that developers often “self-admit” the technical debt by inserting comments indicating that the code is temporary and will need to be reviewed in the future. Furthermore, after a manual analysis of the code, they identify 62 different comment schemes indicating SATD. They showed that in software projects, SATD is very common, and that it is introduced mainly by experienced developers. Liu et al. [11] propose a SATD Detector, which is a tool that can automatically detect SATD comments using text mining and highlight, list, and manage detected comments in an integrated development environment (IDE). This tool consists of a Java library and an Eclipse plug-in.

Maldonado and Shihab [12] have developed an approach that allows identifying SATD instances in the code through comments posted by the developers. The proposed approach is based on model matching and classifies the SATD into five types: design, defect, documentation, requirements, and tests.

Maldonado et al. [13] have also surveyed developers involved in the introduction and/or removal of technical debt. They have found that SATD is predominantly removed when bugs are fixed or new features are added, and that it is removed by the same person that introduced it.

Zampetti et al. [8] conducted a depth quantitative and qualitative study to understand how SATD is removed in the source code. On the one hand, they assessed whether the SATD is “accidentally” removed, and on the other hand to what extent the removal of the SATD is documented. Therefore, they have deepened the study of the relationship between the removal of comments that document SATD and the related changes to the source code, highlighting through their results that a large percentage of removal of SATD comments occurs “accidentally” when the whole method is removed. Furthermore, the removal of SATD is documented in commit messages in only 8% of cases.

In addition, Iammarino et al. [14], analyzed whether the refactoring operations coincide with the removal of SATD to understand if it is a cause–effect relationship or a randomness. They found that

there is a greater chance that refactoring will take place where there has been a removal of the SATD, but that the two operations are unrelated because sometimes, the changes affect different parts of code.

The proposed study differently from the aforementioned research studies focuses on the measures of technical debt to understand if and how they are related to each other. In particular, this study starts from the results of Zampetti et al. [8], because we used their dataset consisting of a set of methods labeled with SATD at the design level and the commit ID in which the SATD was removed.

## 2.2. Investigation of Effects on SATD and Quality Metrics

In the literature, there are numerous studies about the relationships between SATD and quality metrics. In [15], the authors analyzed how technical debt and software quality are related. In particular, they considered the defects occurrences in files with self-admitted technical debt respect to files without self-admitted technical debt to investigate the possibility that differences of the debt lead to the introduction of defects.

In [16], Griffith et al. reported about the relation between some relevant software quality metrics (such as, Weighted Methods per Class, Coupling between Object Classes, and Readability) and SATD.

More recently, in [17], the authors described an analysis aimed at studying to what extent self-admitted technical debt and software quality metrics are related. From their results, it emerged that self-admitted technical debt will remain for extensive time in the source code and is often widespread.

The results of this study show that although technical debt can have negative effects, its impact is not related to defects; rather, it makes the system more difficult to modify in the future.

For more detail on these aspects, in [5], an empirical study is reported about the SATD removals, investigating the time they remain in the software project and the person removing them. Their results confirmed that mainly the self-admitted technical debt is self-removed, but in some cases, it takes a long time to be removed, which often occurs due to bug fixes and does not follow a specific removal process. More recently, other researchers focusing on developing approaches for TD automated documentation or automated suggestions for TD removal [18] must also focus on other actions that change the program's behavior, e.g., changing Application Programming Interfaces or pre and post conditions. In [19], the authors investigate the possibility of using SATD comments to resolve architectural divergences. They leverage a dataset of previously classified SATD comments to trace them to the architectural divergences of a large open source system, namely ArgoUML.

## 3. Study Setup

The aim of this work is to investigate the different ways of referring to technical debt, comparing subjective approaches with a more objective measure made on the source code of the software. The following subsections contain the description of the research questions and the process used for the extraction of data useful for the study.

### 3.1. Research Questions

This paper reports on the effects of removing the SATD, on the objective measurement of technical debt, and other metrics of the main source code, in open source software projects. The study is based on the following research questions:

**RQ1:** *To what extent do self-admitted technical debt removals actually lead to lower technical debt value?*

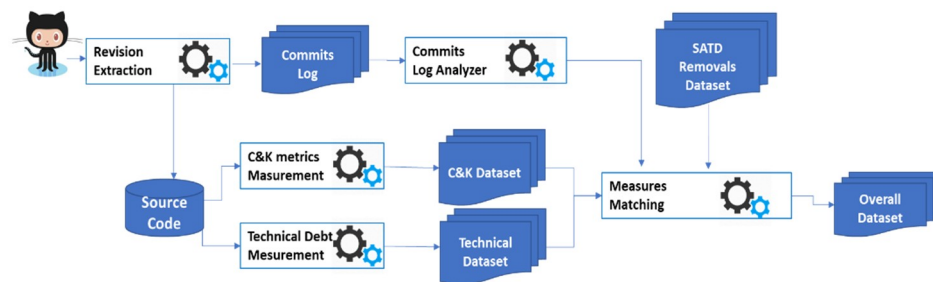
This research question aims to understand if self-admitted technical debt removals correspond to lower values of the technical debt measure. To perform the assessment, a fine-grained analysis has been performed, collecting and analyzing the data of open-source software systems, commit per commit, and class per class. Descriptive statistics and graphs have been used to support the analysis.

**RQ2:** *To what extent do self-admitted technical debt removals lead to lower Chidamber and Kemerer metrics values?*

The goal of this research question is to look for empirical evidence to support the hypothesis about the positive impact of SATD removals on software quality metrics. In particular, the study focused on the OO metrics with the objective of understanding if their values improve when SATD removals occur.

### 3.2. Data Extraction

Figure 1 shows the toolchain used in this analysis, highlighting the tools adopted, and the input provided, and the output obtained.



**Figure 1.** Process used to conduct the analysis.

As shown in the figure, to keep track of all the source code changes made to each file, the first step was to analyze the source code repository. To this end, all changes made to the files have been extracted and stored in a dataset. The SonarQube tool, an open-source platform capable of providing information on the quality of a software system, was used to extract the TD value for each file.

SonarQube is based on using the Software Quality Assessment based on Lifecycle Expectations method to measure code quality [20] and provides a dashboard for viewing results.

In particular, in this study, SonarQube has been used only for the TD measure, which quantifies in minutes the effort in working time to fix existing issues in a file. The SonarQube Web API was used to extract the value of the technical debt, using an http request query: [http://localhost:9000/api/measures/component=projectidcommitid&metricKeys=sqale\\_index](http://localhost:9000/api/measures/component=projectidcommitid&metricKeys=sqale_index).

OO metrics have been assessed using the CK tool [21]. More in detail, its functionality has been integrated into the java code developed for analysis. In particular, CK calculates the class-level and metric-level code metrics in Java projects using static analysis. Its analysis is performed on the source code and not on the compiled code. In particular, it evaluates a large set of class-level metrics, including the widely used CK class-level metrics. Specifically, the metrics included in the dataset are listed in Table 1.

As previously pointed out, we have measured the trend of all CK metrics (the working dataset with all metrics collected and analyzed is available for replication purposes ([https://drive.google.com/drive/folders/1k9wcjAus\\_wYmGSuGhvkq7qY3Tk6Uexht?usp=sharing](https://drive.google.com/drive/folders/1k9wcjAus_wYmGSuGhvkq7qY3Tk6Uexht?usp=sharing))). In detail, for the metrics with consolidated threshold values defined in the literature [22], which are lack of cohesion of methods (LCOM), Depth of Inheritance Tree (DIT), Public fields, and Public methods, the analysis has been performed respect to those thresholds. Meanwhile, all the other metrics values are compared to their previous values to establish if they were improved or not.

**Table 1.** Chidamber & Kemerer metrics included in the dataset.

Metric	Description
CBO	Coupling between objects: a total of the number of classes that a class referenced plus the number of classes that referenced the class. If a class appeared in both the referenced and the referred classes, it was only counted once.
DIT	The Depth of Inheritance Tree (DIT) measures inheritance levels from the hierarchy of objects above, so it is the maximum length of a path from a class to a root class in a system's inheritance structure. Measure how many superclasses can affect a class. For a class, its minimum value is 1.
Number of fields	Counts the number of fields. Specific numbers for the total number of fields, static, public, private, protected, default, final, and synchronized fields.
NOSI	Number of static invocations: Counts the number of invocations to static methods. It can only count the ones that can be resolved by the Java Development Tools.
RFC	Response for a class: Shows the interaction of the class's methods with other methods, thus the total number of methods that can potentially be executed in response to a message received from an object of a class.
WMC	Weight method class or McCabe's complexity. It counts the number of branch instructions in a class.
LOC	Lines of code: It counts the lines of code, ignoring empty lines. The number of lines here might be a bit different from the original file, as the JDT's internal representation of the source code is used to calculate it.
LCOM	Lack of cohesion of methods: measures the correlation within a class between local methods and instance variables. If there is a high cohesion, it means that there is a good division; on the contrary, the lack of cohesion or low cohesion follows an increase in complexity. In this case, the solution is represented by the subdivision of this class into several subclasses.
Public Fields	Counts the total number of public fields defined in a class. Publics fields refer to an object that is directly accessible and edited by other objects. Therefore, its use can cause a strong coupling between the classes within a software system, reducing the modularity of the program.
Public Methods	Find the total number of public methods defined in a class. This metric can be considered as an indicator of how large a class is, so it represents the number of features that the class provides.

### 3.3. Linking SATD to Technical Debt Values

Moving from the SADT removals dataset [8], considering the list of commits where the SATD removals occur and the temporally previous commits, a new dataset has been constructed measuring for each of these commits the technical debt values, and the Chidamber and Kemerer metrics values, as previously explained. Once this new dataset has been obtained, it was necessary to perform an inner join between them. This was automatically obtained by a tool implementing the following Algorithm 1.

In the end, the dataset obtained for the analysis includes both the data on the clones and the data on the technical debt.

### 3.4. Subject Projects

The study reports results involving four Java software projects: Log4j (<https://github.com/apache/log4j>), Gerrit (<https://github.com/GerritCodeReview/gerrit>), Hadoop (<https://github.com/apache/hadoop>), and Tomcat (<https://github.com/apache/tomcat>), which are different in size, number of commits, and application domains. The choice fell on these systems because their programming language is Java, they still have an active Git repository containing multiple versions, and there are variations in the application domains, sizes, and revisions. Table 2 details for each system the versions analyzed, the branch numbers, the number of commits, and the total number of commits in which at

least one SATD Removal has been identified. Commits with SATD removals were identified by the Maldonado et al. study [13], which created a dataset with an analysis that began on 15 March 2015.

**Table 2.** Systems analyzed.

System Projects	Branches	Commits	SATD Removal
Log4j	7	14.296	37
Gerrit	15	318.362	61
Hadoop	274	2.721.039	154
Tomcat	4	22.215	302

**Algorithm 1:** Matching commits (SATD Removals  $\Rightarrow$  TD values)

---

```

1.   Input: C: Commits Sets, D: SATD Dataset
2.   Output: Dataset (CSV format)
3.   for all  $d \in D$  do
4.     O: set of commits hash to be analyzed
5.      $r \rightarrow$  hash of removal commit in  $d$ 
6.      $previous \rightarrow$  retrieve hash of previous commit of  $r$  from C
7.      $next \rightarrow$  retrieve hash of next commit of  $r$  from C
8.     push  $r, previous, next$  in O
9.     for all  $o \in O$  do
10.      clone repository at status of  $o$ 
11.       $file \rightarrow$  file to analyze
12.      if  $file$  exists then
13.        generates.  $properties\ file$ 
14.        executes sonar scanner analysis
15.        recovers TD with sonar web API
16.        deletes.  $properties\ file$ 
17.        run CK analysis
18.        recovers CK object related to  $file$ 
19.      else if  $o$  is equal to  $r$ 
20.        write into the output file "-" for TD, delta and CK metrics
21.      continue
22.      end if
23.      restore repository at current state
24.    end for
25.    if analyses of  $previous$  or  $next$  commit are equal to null then
26.      write into the output file "-" for TD, delta and CK metrics
27.      continue
28.    end if
29.    calculates the delta between the commit removal and the previous one
30.    and between the next commit and the commit removal
31.    write into the output file all commit attributes relating to the analyses performed
32.  end for

```

---

## 4. Results

In this section, we summarize the analysis made and the obtained results.

### 4.1. RQ1: To What Extent Do Self-Admitted Technical Debt Removals Actually Lead to a Lower Technical Debt Value?

To answer this research question, the delta of the TD value is measured by comparing its value at the commit removal and the one at the previous commit. Then, the trend of TD has been analyzed to



obtain the percentages of cases in which the TD measure got worse, improved, or remained unchanged. The case in which it was not possible to provide a measure, due to the absence of the files at the commit analyzed, has also been considered.

Furthermore, the stability of the decreasing trend of the TD value has also been investigated, comparing the value at the commit removal with the one at the successive commit. Specifically, the number of cases where an improvement of the TD in correspondence of a commit removal is propagated also to the next commit has been computed. Therefore, the number of successive commits with a value of the TD equal to or lower with respect to the commit removal has been identified. Indeed, these cases could correspond to a positive action of the developers on the same metric.

Finally, an analysis to observe the impact of the SATD removals on the absence of some files in correspondence with the commits analyzed has been carried out.

#### 4.1.1. Change in TD Value between Commit Removal and Previous Commit

For each analyzed system, Table 3 shows, in the first column, the type of delta between the commit removal and the previous commit, and in the second column, the number of files analyzed.

**Table 3.** Previous commit analysis vs. commit removal.

Delta	# of Files			
	Log4j	Gerrit	Hadoop	Tomcat
File not found	29	32	142	235
Unchanged	4	17	20	87
Increased	10	9	24	54
Decreased	20	13	19	100

The change in the technical debt values, delta, is distinguished in 4 types of possible change observed: file not found, unchanged, increased, and decreased.

In the case of Log4j, Table 3 shows that for 46.03% (29 files), the variation in the TD could not be assessed, as the analyzed file was absent in at least one of the commits to be analyzed.

However, for 31.75% of cases (20 files), there was a decrease in the value of the TD between the previous commit and the commit removal considered, demonstrating that effectively, the changes performed by the software developers lead to a lower value of the technical debt objectively measured on the classes subject to the interventions. In particular, the effect of this reduction is not trivial; indeed, the intensity of change observed in the TD value compared to one of the previous commits ranges from 2 to 80%.

However, in 6.35% of cases (4 files), no change in the analyzed value was found.

Finally, for the remaining 15.87% of cases (10 files), the TD values increase. These results are not expected, as it means that the percentages indicate that the changes in the source code made to perform a SATD removal do not always lead to a more objective reduction of the TD. More specifically, in the commits analyzed, there was an intensity of increase in the TD value that ranges between 2% and 100%, with a particular case where the increase is even equal to 173% because the TD value goes from 15 to 41 min.

In Gerrit, it can be observed that in 45.07% of cases (32 files), it was not possible to measure the variation of the TD due to the absence of the file with SATD in the commit. However, for 23.94% of the cases (17 files), there was no change in the TD, and for 12.68% of the cases (13 files), there was a deterioration in the value. The worsening recorded corresponds to an increase in the TD value between commits, which fluctuates in a range from 4 to 81%. Only in the remaining 18.31% of cases (9 files) does the measure of the TD of the commit removal get an improvement compared to the previous commit. In these commits, there was a decrease in the TD value ranging between 6% and 81%.

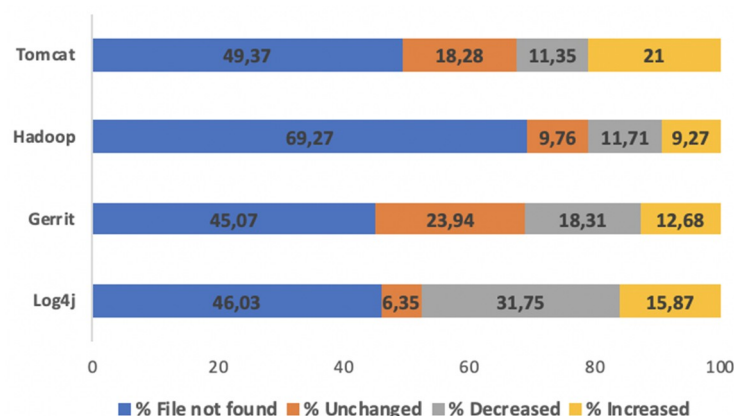
Even for Hadoop, there is a high percentage of cases, 69.27% (142 files), in which it was not possible to measure the variation of the technical debt due to the absence of the file to be examined in correspondence with one of the commits to be analyzed. However, for 9.27% of the cases (20 files), there was no change in the TD, in connection with the commit removal, and for 9.76% of the cases (19 files), there was a worsening of the value. In fact, there was an increase in the value of TD which varies between 0.9% and 197%. In fact, in several cases, the value of the TD between a commit and the next commit increased more than double. Two cases have even been identified in which the value increased by 400% and 500%, respectively from a value of 1 to 6 min, and 5 to 25 min. Only in the remaining 11.71% of cases (24 files) does the measure of the TD of the commit removal improve compared to the previous commit. In these cases, the decrease varied between 0.28% and 97%. These percentages suggest that only in a few circumstances are the changes made by maintainers to resolve the TD valid, while in most cases, the performed solutions do not lead to an effective decrease of the metrics values.

As for the previous systems, also in the case of Tomcat, there is a high percentage of cases, 49.37% (235 files), for which it was not possible to measure the TD due to the absence of the files.

In 21% of cases (100 files), there was a decrease in the TD value compared to the previous commit, while in 11.35% (54 files), there was an increase in the value. Respectively, in the cases analyzed, the decrease varied between 0.19% and 100%, while the increase varied between 0.2% and 120%. Outliers were also identified in the increase; in fact, in one case, the TD increased by 800%, going from 2 to 18 min, in another by 346%, changing from 26 to 111 min.

Finally, 18.28% (87 files) of the times, there was no change in the value of the TD between the analyzed commit and its previous one.

Figure 2 depicts an overview of the delta trend for each project, highlighting how in most cases, the file is not found (blue bars), compared to the other cases, in which the TD value is unchanged (orange bars), decreased (gray bars), and increased (yellow bars).



**Figure 2.** Overview of the change type percentage between the previous commit and the removal commit for each project.

Overall, the data obtained are different with respect to the expectations. It clearly emerged that there is a high subjectivity in the processes leading to the identification and removal of the SATD. In many cases, the SATD removals do not reflect on similar results from an automated evaluation process, such as the objective measure provided by SonarQube. Undoubtedly, this is due to different approaches used by developers and maintainers to implement the different solutions, but the number of not aligned cases suggests at least the need for rethinking the subjective process on SATD evaluations.



#### 4.1.2. Change in TD Value between Commit Removal and Subsequent Commit

Differently from the previous case, this analysis considers the commit of the SATD removal and its consecutive. For each system analyzed, Table 4 shows, in the first column, the type of delta trend between the commit removal and the consecutive commit, and in next columns, the number of files found for each projects.

**Table 4.** Removal commit analysis vs. consecutive commit.

Delta	Number of Files				Delta
	Log4j	Gerrit	Hadoop	Tomcat	
File not found	29	32	142	235	File not found
Unchanged	27	34	54	222	Unchanged
Increased	5	3	5	8	Increased

Table 4 shows that for Log4j in 46.03% of cases (29 files), the file has been removed, while in 42.86% of cases (27 files), the TD remains unchanged between subsequent commits. For 7.94% of cases (5 files), there was an increase in the value of the metric under examination, while for the remaining 3.17% (2 files), a decrease in value was found. Specifically, the increase in TD changes in a range from 0.6 to 8%, while the decrease is between 25% and 50%.

In Gerrit, as in the previous case, there is a high percentage of cases 45.07% (32 files) where the file has been removed. The table points out that in 47.89% of cases (34 files), the TD value remains unchanged in subsequent commits. Instead, for 4.23% of cases (3 files), there is an increase in the value of TD, while just 2.82% (2 files) correspond to a decrease in value. In the commits analyzed, we observed the intensity of increase ranging from 5 to 48%, instead of the decrease from 7 to 40%.

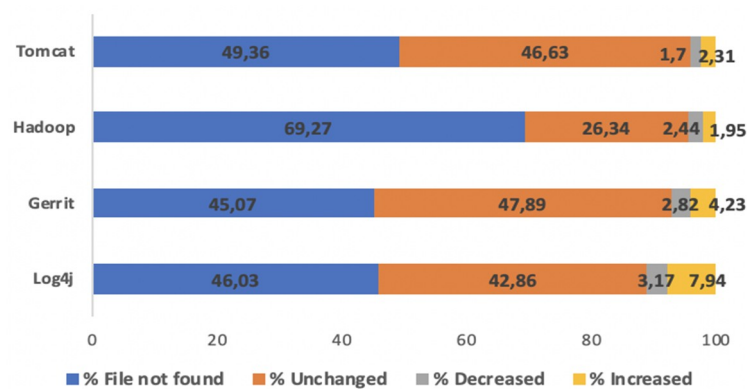
In Hadoop, there is the highest percentage of files not found, 69.27% (142 files); in 26.34% of cases (54 files), the TD value remains unchanged in subsequent commits. For 1.95% of the cases (5 files), an increase in the value of the TD was found. Tomcat is the only system in which the percentage of the increase varies in a much wider range, more in detail, between 0.27% and 160%, with a case in which it even reaches 800% because it varies from 3 min to 27 min. Instead, for 2.44% (4 files), a decrease in the value was found, with a percentage of change that varies between 3% and 9%.

As for Gerrit also in the case of Tomcat, the highest percentages refer to the number of cases for which the files are not found, 49.36% (235 files), and there were no changes in the value of the TD between the commit analyzed and its subsequent, 46.63% (222 files).

For 1.7% (8 files), there is an increase in the TD value with a percentage of change that varies between 0.23% and 21%, while for 2.31% (11 files), there is an improvement in the TD value, which is recorded as a decrease in the value itself. This decrease in the value varies between 0.81% and 67%.

Overall, these results confirm the ones obtained in the previous analysis and show that the impact on the TD values is the maintained in subcommits. Then, if the removal of SATD leads to a change in the TD values, this can be observed even in the subsequent commits.

Figure 3 highlights that the highest percentages concern cases of files not found, followed by unchanged, unlike the previous analysis, where the highest percentage was recorded in the decrease in the TD value. These results suggest that in most cases, developers tend not to reevaluate their modification choices once they have intervened on a certain class, leaving the code for subsequent commits unchanged or modifying it marginally.



**Figure 3.** Overview of the change type percentage between the removal commit and the subsequent commit for each project.

#### 4.1.3. Preservation of the Trend between Negative Delta Commit Removal and Next Commit

After identifying the commit removals for which there was a decrease in the value of the TD (negative Delta) compared to the previous commit, we proceeded to count for how many of these the subsequent commit preserved the improvement.

For each system analyzed, Table 5 shows the number of cases in which the trend is preserved or not in the next commit.

**Table 5.** Negative delta.

Delta	# of Files			
	Log4j	Gerrit	Hadoop	Tomcat
Preserved trend in consecutive commit	16	11	21	185
Not preserved trend in consecutive commit	4	2	3	7

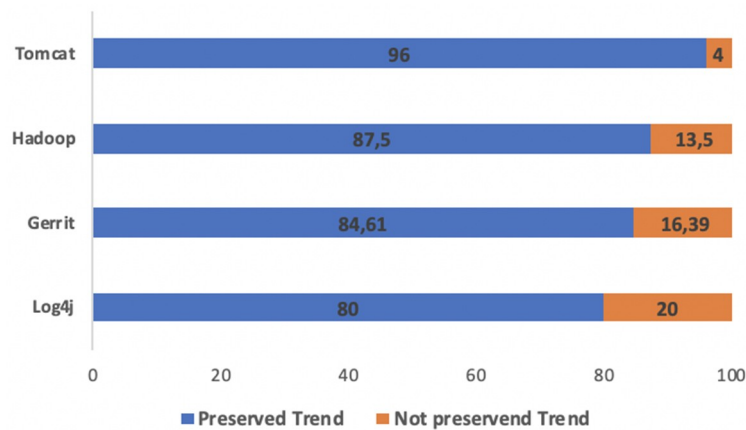
In log4j there are 16 cases, equivalent to 80%, in which the TD value was lower or otherwise unchanged, while for the remaining 4 cases, an increase equivalent to 20% was found.

For Gerrit, in 85% of cases (11 files), the TD value for the next commit remained unchanged or decreased compared to the commit removal, and for 15% of cases (2 files), the value of the TD gets worse in the next commit.

The table also highlights that for Hadoop, in 87.5% of cases (21 files), the value of the TD relating to the next commit remained unchanged or further improved, while in 12.5% of cases (3 files), the value TD has worsened since commit removal.

For Tomcat, the results are similar, because in 96.35% of cases (185 files) the trend is preserved, instead of in 3.65% of cases (7 files), where there is a worsening of the TD value, and therefore the trend is not preserved.

Finally, Figure 4 depicts the percentage of the trend of the TD value, comparing the set of cases in which the trend is preserved (blue bars), with the other cases in which the trend is not preserved (orange bars). As the figure shows, the percentage is substantially higher—about four times higher—for preserved trend than for not preserved. It means that developers tend not only to maintain the quality of the code but also to try to improve it, following positive actions on the classes.



**Figure 4.** Overview of the trend preservation percentage between negative delta commits removal and the next commit for each project.

#### 4.1.4. Relationship between Change Type of Commit Removal and File not Found

Considering the 4 types of changes that can be made to remove the SATD, which are listed in Section 1, the following Table 6 shows, for each project, the number of files not found, the changes that had been applied to these files, and the relative percentage.

**Table 6.** Commits removals.

Change Type	Number of Files			
	Log4j	Gerrit	Hadoop	Tomcat
Class Removal	28	14	54	227
Method Removal	1	14	16	1
Method Changed	0	3	62	6
Method Unchanged	0	1	10	1

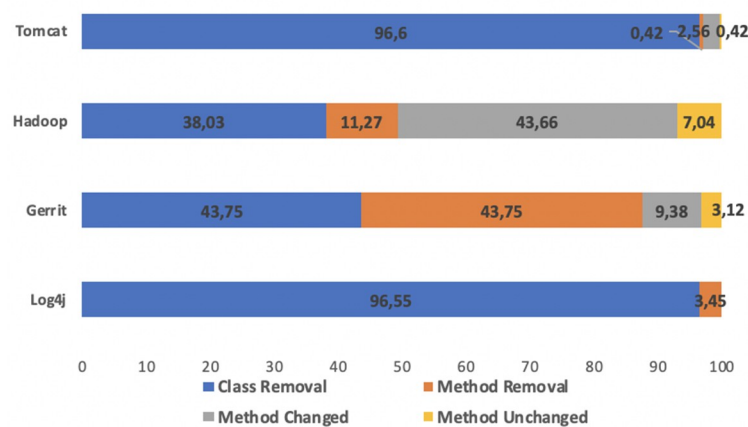
In Table 6, it is possible to observe that for Log4j, in 96.55% of cases (28 files) when the file was absent, this was determined by a Class Removal operation, where the class to be analyzed was deleted. However, for the remaining 3.45% of cases (1 file), the lack of the file was due to a Method Removal operation.

For Gerrit, there is the same percentage of cases, 43.75% (14 files), where the file is not found, and the removal is due to the removal of a method or class. Unlike log4j, it has been found that there are some cases for which there is a change of method 9.38% (3 files), and the method is unchanged 3.12% (1 file).

Hadoop represents the only case in which the absence of the files is due to the change method with a percentage of 43.66% (62 files). The removal of the class follows with 38.03% (54 files), followed by the removal of the method 11.27% (16 files) and the unchanged method 7.04% (10 files).

Finally, Tomcat follows the same trend as Log4j and Gerrit, because the highest percentage, 96.60% (227 files) occurs in the case of class removal, which is followed by 2.56% (6 files) Method Changed and 0.42% (1 file) for both Method Removal and Method Unchanged.

Figure 5 shows that in Log4j, Hadoop, and Tomcat, the most used type of modification to remove the TD, when the file is not found, is the removal of the class. Instead, in the case of Gerrit, in addition to the removal of the class, the method is removed as well. So, it means that in general, developers tend to completely remove the file or method with the TD inside.



**Figure 5.** Overview of the change type percentage when the file is not found.

#### 4.2. RQ2: To What Extent Do Self-Admitted Technical Debt Removals Lead to Lower Chidamber and Kemerer Metrics Values?

To understand to what extent the metrics of Chidamber and Kemerer vary following the removal of the technical debt, the change in the value of the TD was evaluated concerning the CK metrics. Then, concerning a subset of metrics (lack of cohesion in method, depth of inheritance tree, public fields, and public methods), their fit with available thresholds was analyzed.

We compared the TD value trend between two successive commits with the values of CK metrics. In particular, we verified whether a decrease in the TD corresponded to an improvement in most of the metrics listed above. On the other hand, we investigated even if a decrease in the TD value was associated with a worsening of most of the metrics.

Specifically, if the developers' removal of the SATD did not produce improvements for the TD, we evaluated its effect on the other quality metrics. In particular, if a non-improvement of the technical debt (worsening or unchanged of the TD) found an improvement in most of the metrics, the action of the developers had a beneficial side effect for the system, despite a negative impact on the TD. If, on the contrary, the non-improvement of the TD is also associated with a worsening of most of the metrics considered, then a completely pejorative effect has occurred on the system. The last case (NA) is one in which the value of all the metrics described above remained unchanged, or there was an equal number of improved and worsened metrics, or again, there was no change in the value of the TD among the two successive commits; therefore, it was not possible to catalog.

For each project, Table 7 reports the number of cases in which (1) the TD has improved and the metrics have improved, (2) the TD has improved and the metrics have deteriorated, (3) both TD and metrics have deteriorated, and (4) the TD has deteriorated, but the metrics have improved, and (5) other cases not considered.

**Table 7.** Metrics analysis. TD: technical debt.

Project	#Files TD Improved & Metrics Improved	#Files TD Improved but Metrics Got Worse	#Files Both TD and Metrics Got Worse	#Files TD Got Worse but Metrics Improved	#Files NA
Log4j	13	6	12	1	36
Gerrit	7	6	10	10	45
Hadoop	12	2	13	11	88
Tomcat	38	15	11	12	135

For Log4j, for 19.12% of the cases (13), the result produced by SonarQube was validated, since an improvement in the Technical Debt measure was matched by an improvement in the majority of the metrics taken into consideration. For 8.82% of the cases (6), the result was not validated, as an

improvement in the value of the TD was accompanied by a worsening of most of the metrics considered. However, for 1.47% of cases (1), the non-improvement of the TD value had a positive side effect, improving most of the metrics considered. However, for 17.65% of the cases (12), in addition to the non-improvement of the TD, there was also a worsening of the majority of the metrics taken into consideration. Finally, for 52.94% of cases (36), it was not possible to make considerations.

For Gerrit, in most cases, 12.82% (10 cases) of developers' actions do not cause an improvement in the TD but hurt both the TD and the metrics. With the same percentage, there is a worsening of the TD, but at the same time, there was a considerable improvement of the metrics. Only in 8.97% of cases (7) did the results show an improvement of both values under observation, while in 7.69% of cases (6), there was an improvement in TD but a worsening of metrics. For the remaining 57.69% of cases (45), it was not possible to express considerations. Therefore, it is observed that the cases in which conclusions cannot be drawn represent a significant percentage.

For Hadoop, the table shows that for 9.52% of cases (12), an improvement in technical debt was accompanied by an improvement in at least one of the metrics considered. In 1.59% of cases (2), an improvement in the value of TD was accompanied by deterioration in at least one of the metrics considered, and none of them improved.

For 8.73% of cases (13), the developers' actions did not lead to any improvement in the value of the TD but improved most of the metrics considered, thus obtaining a positive effect on them. For 10.32% of cases (11), there was not an improvement in TD combined with a worsening of the metrics examined. Finally, in 69.84% of cases (88), it was not possible to take these factors into consideration.

Tomcat also confirms the trend of the previous cases; in fact, the highest percentage belongs to the case where no consideration was possible, 64% (135 cases).

This percentage was followed with 18% (38) cases in which both the TD value and metrics improved, and with 7.10% of cases (15) in which there was an improvement in the TD value and at the same time a worsening of the metrics. Finally, in 5.69% of cases, (12) there is a worsening of both values, and for 5.21% of cases (11), the TD worsens but the metrics improve.

Figure 6 depicts the data about the trend of the TD value and the CK metrics. The Figure shows that for Log4j, the cases with SATD removals correspond to improved metrics and improved TD in more than half of the cases (52.95%). For Gerrit, the number of cases where there is an improvement is roughly equal to the number of cases where this has not happened. We have also noticed how the cases in which the developer's activity produces negative effects on TD values, but positive effects on the metrics, are almost the same as those in which all the metrics are negatively affected.

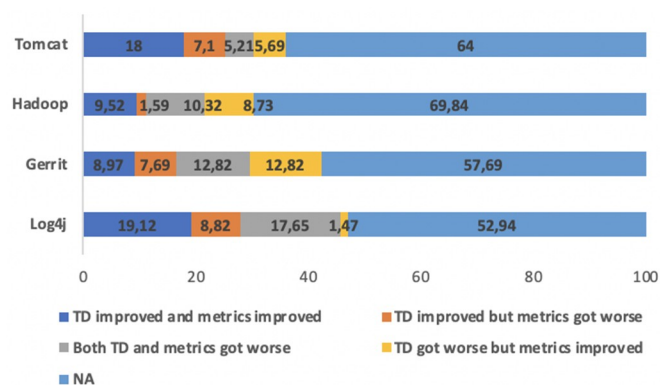


Figure 6. Overview of the trend of the TD value and the CK metrics.

Finally, unlike the previous project, for Hadoop and for Tomcat, it is possible to notice a greater number, and therefore a great prevalence, of cases in which it is not possible to make assumptions about the relationship between TD and metrics. However, for Hadoop, there is only a small percentage for which there is an improvement in the TD and a worsening of the metrics, and the number of cases

in which the action of the developers led to a non-improvement of the technical debt is approximately equal to a positive or negative effect on the metrics considered. Instead, in Tomcat, it is clear that the number of cases in which an improvement in the TD has also led to an improvement in the metrics is more than two times higher than the other cases in which the opposite occurs, or the improvement of one and the worsening of the other.

## 5. Threats to Validity

The findings of the research proposed are subject to the following threats to the validity.

Construct Validity is related to the extent to which the technical debt measures used are actually reliable. To deal with this issue, the technical debt values have been evaluated using SonarQube tool. Actually, SonarQube is a widely used open-source tool allowing the identification and the evaluation of technical debt.

The internal validity threat concerns whether the results obtained correctly follow from the collected data—specifically, whether the metrics are significant to our conclusions and whether the evaluations are suitable. In this study, the use of SonarQube to measure Technical Debt [20] has been related to self-admitted technical debt [10] to assess human-based evaluation with respect to the automatic assessing of technical debt.

The external validity threat refers to the possibility of generalizing the obtained results. Indeed, the size of selected dataset is smaller with respect to the population of open source projects; therefore, this could impact the generalizability of our conclusions. To mitigate this threat, well-known software projects have been considered, which are continuously evolving and different for different dimensions, domain, size, timeframes, and the number of versions. Nevertheless, several limitations to the generalizability of the conclusions remain.

## 6. Conclusions

In this study, the relationship between SATD removal and TD values measured has been investigated using objective criteria. By leveraging a dataset of SATD and their removals in four open-source projects and by using the SonarQube tool for measuring the TD values, the trends of TD values when SATD removals occur have been analyzed. Overall, the results of the study indicate that SATD removals in few cases correspond to an effective reduction of TD values, while in numerous cases, the classes are indicated as SATD removals disappear from the repository. In particular, from the obtained results, it clearly emerges that in about half of the cases, the files are removed when SATD removals occur for all the four systems analyzed. Indeed, SATD removals correspond to class removal operations in 59.44% of cases and method removal in 19.49% of cases. Intuitively, this may indicate that developers tend to face technical debt resolution problems by eliminating the classes involved. This is a strong solution from a software design point of view and suggests that it is possible to hypothesize that the methods and functions belonging to the removed classes can be moved and integrated into other classes. The Method Changed corresponds to 17.68% of cases and Method Unchanged represents the remaining 3.39%.

Regarding the delta of the technical debt between commit removals and the following commits, in 20.59% of cases, there is a negative value, while it increases or remains unchanged respectively for 12.61% and 13.35% of the cases.

These results indicate that self-admitted technical debt by developers and its removal are too subjective, and this can lead to very different results from measurements made by means of objective tools.

In the cases where commit removals correspond to a positive decrease of the technical debt, it is possible to note that in 84.04% of cases, the next commit preserves the improvement, exhibiting the same value of TD or the lower one. Then, developers tend to adopt strategies and carry out activities aimed at preserving the results obtained through a specific commit removal. This can be observed in the bar charts relating to the difference in the value of the Technical Debt between commits removal



and its subsequent commit. In the graph, the highest percentage, excluding cases of class removals, corresponds to the preservation of the TD value.

The study also considered the relationship with the OO metrics measured with the CK tool. In 12.54% of cases, it emerged that an improvement in the measure of the technical debt corresponds to an improvement of the part of metrics taken into consideration. In 6.03% of cases, the results are not aligned, and the improvement in the value of the TD corresponds to the worsening of most of the metrics considered. Finally, in the 7.67% of the cases, the not improved TD value has a positive effect, improving most of the metrics considered. However, in 13.60% of cases, it can be observed that the worsening of the TD also corresponds to the worsening of many of the metrics considered.

Overall, the results indicate that self-admitted technical debt is not adequately used by the software developer. Indeed, their removals actually do not have a significant effect on the quality of the metrics' values.

The results obtained will be used to address future research studies to improve the analysis about the technical debt objective and subjective measures. In the future, we envision to carry out a quantitative and qualitative investigation on the effects on quality profiles of software project deriving from the use of subjective measure of technical debt. Of course, in future work, more software projects will be considered with different characteristics, such as different programming languages and application domains. Moreover, the relationships with quality attributes will be extended to perform an in-depth quantitative and qualitative analysis.

**Author Contributions:** Conceptualization, L.A., M.I.; Methodology, L.A., M.I.; Formal Analysis Investigation M.C., A.D.V., L.N.; Data curation M.C., A.D.V., L.N.; Writing—original draft preparation, L.A., M.I., M.C., A.D.V., L.N.; Writing—review and editing, L.A., M.I. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Conflicts of Interest:** The authors declare no conflicts of interest.

## References

1. Aldecoa, R.; Marín, I. Exploring the limits of community detection strategies in complex networks. In Proceedings of the 2004. Proceedings. 20th IEEE Metrics-based rules for detecting design flaws International Conference on in Software Maintenance, Chicago, IL, USA, 11–12 September 2004; IEEE: Piscataway, NJ, USA, 2013.
2. Wong, S.; Cai, Y.; Kim, M.; Dalton, M. Detecting software modularity violations. In Proceedings of the Proceeding of the 33rd international conference, Association for Computing Machinery (ACM), Granada, Spain, May 2011.
3. Li, Z.; Liang, P.; Avgeriou, P.; Guelfi, N.; Ampatzoglou, A. An empirical investigation of modularity metrics for indicating architectural technical debt. In Proceedings of the 10th international ACM Sigsoft conference on Quality of software architectures (QoSA '14). Association for Computing Machinery, New York, NY, USA, 27 June 2014; pp. 119–128. [\[CrossRef\]](#)
4. Farias, M.A.D.F.; Neto, M.G.D.M.; Da Silva, A.B.; Spinola, R.O. A Contextualized Vocabulary Model for identifying technical debt on code comments. In Proceedings of the 2015 IEEE 7th International Workshop on Managing Technical Debt MTD, Bremen Germany, 2 October 2015; pp. 25–32. [\[CrossRef\]](#)
5. Maldonado, E.D.S.; Shihab, E.; Tsantalis, N. Using Natural Language Processing to Automatically Detect Self-Admitted Technical Debt. *IEEE Trans. Softw. Eng.* **2017**, *43*, 1044–1062. [\[CrossRef\]](#)
6. Brondum, J.; Zhu, L. Visualising architectural dependencies. In *2012 Third International Workshop on Managing Technical Debt MTD*; IEEE: Piscataway, NJ, USA, 2002; pp. 7–14. [\[CrossRef\]](#)
7. Li, Z.; Liang, P.; Avgeriou, P. Architectural Technical Debt Identification Based on Architecture Decisions and Change Scenarios. In Proceedings of the 2015 12th Working IEEE/IFIP Conference on Software Architecture Institute of Electrical and Electronics Engineers (IEEE), Victoria, BC, Canada, 29 September 2015; pp. 65–74.
8. Zampetti, F.; Serebrenik, A.; Di Penta, M. Was self-admitted technical debt removal a real removal? In Proceedings of the 15th International Conference on Computer Systems and Technologies—CompSysTech '14 Association for Computing Machinery (ACM), Gothenburg, Sweden, 27 May 2018; pp. 526–536.

9. Chidamber, S.; Kemerer, C. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.* **1994**, *20*, 476–493. [\[CrossRef\]](#)
10. Aniket Potdar and Emad Shihab. An Exploratory Study on Self-Admitted Technical Debt. In Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME '14). IEEE Computer Society, Washington, DC, USA, December 2014; pp. 91–100. [\[CrossRef\]](#)
11. Liu, Z.; Huang, Q.; Xia, X.; Shihab, E.; Lo, D.; Li, S. SATD Detector: A Text-Mining-Based Self-Admitted Technical Debt Detection Tool. In Proceedings of the ICSE 2018, DEMO—Demonstrations, Gothenburg, Sweden, 30 May 2018.
12. Maldonado, E.D.S.; Shihab, E. Detecting and quantifying different types of self-admitted technical Debt. In Proceedings of the 2015 IEEE 7th International Workshop on Managing Technical Debt MTD, Bremen, Germany, 2 October 2015; pp. 9–15. [\[CrossRef\]](#)
13. Maldonado, E.D.S.; Abdalkareem, R.; Shihab, E.; Serebrenik, A. An Empirical Study on the Removal of Self-Admitted Technical Debt. In *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*; Institute of Electrical and Electronics Engineers (IEEE): Piscataway, NJ, USA; pp. 238–248.
14. Iammarino, M.; Zampetti, F.; Aversano, L.; Di Penta, M. Self-Admitted Technical Debt Removal and Refactoring Actions: Co-Occurrence or More? In Proceedings of the 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), Cleveland, OH, USA, 30 September–4 October 2019; pp. 186–190. [\[CrossRef\]](#)
15. Wehaibi, S.; Shihab, E.; Guerrouj, L. Examining the Impact of Self-Admitted Technical Debt on Software Quality. In Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Suita, Japan, 14 March 2016; Institute of Electrical and Electronics Engineers (IEEE): Piscataway, NJ, USA; Volume 1, pp. 179–188.
16. Griffith, I.; Reimann, D.; Izurieta, C.; Codabux, Z.; Deo, A.; Williams, B.; Deo, A.; Williams, B. The Correspondence between Software Quality Models and Technical Debt Estimation Approaches. In Proceedings of the 2014 Sixth International Workshop on Managing Technical Debt, Washington, DC, USA, 30 September 2014; pp. 19–26. [\[CrossRef\]](#)
17. Gabriele Bavota and Barbara Russo. A large-scale empirical study on self-admitted technical debt. In Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16). Association for Computing Machinery, New York, NY, USA, 14–15 May 2016; pp. 315–326. [\[CrossRef\]](#)
18. Zampetti, F.; Serebrenik, A.; Di Penta, M. Automatically learning patterns for self-admitted technical debt removal. In Proceedings of the 27th IEEE Inter-National Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, 18–21 February 2020; pp. 355–366.
19. Sierra, G.; Tahmid, A.; Shihab, E.; Tsantalis, N. Is Self-Admitted Technical Debt a Good Indicator of Architectural Divergences? In Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China, 24 February 2019; Institute of Electrical and Electronics Engineers (IEEE): Piscataway, NJ, USA; pp. 534–543.
20. Letouzey, J.-L.; Ilkiewicz, M. Managing Technical Debt with the SQALE Method. *IEEE Softw.* **2012**, *29*, 44–51. [\[CrossRef\]](#)
21. Aniche, M. Java Code Metrics Calculator (CK). Available online: <https://github.com/mauricioaniche/ck>. (accessed on 9 July 2020).
22. Ferreira, K.; Bigonha, M.A.; Bigonha, R.S.; Mendes, L.F.; Almeida, H.C. Identifying thresholds for object-oriented software metrics. *J. Syst. Softw.* **2012**, *85*, 244–257. [\[CrossRef\]](#)

