Hindawi
Scientific Programming
Volume 2020, Article ID 2976564, 14 pages
https://doi.org/10.1155/2020/2976564

Research Article

Comparing Maintainability Index, SIG Method, and SQALE for Technical Debt Identification

Peter Strečanský, Stanislav Chren, and Bruno Rossi

Masaryk University, Brno, Czech Republic

Correspondence should be addressed to Bruno Rossi; brossi@mail.muni.cz

Received 10 January 2020; Revised 26 March 2020; Accepted 16 June 2020; Published 20 July 2020

Academic Editor: Tomas Cerny

There are many definitions of software Technical Debt (TD) that were proposed over time. While many techniques to measure TD emerged in recent times, there is still not a clear understanding about how different techniques compare when applied to software projects. The goal of this paper is to shed some light on this aspect, by comparing three techniques about TD identification that were proposed over time: (i) the Maintainability Index (MI), (ii) SIG TD models, and (iii) SQALE analysis. Considering 20 open source Python libraries, we compare the TD measurements time series in terms of trends and evolution according to different sets of releases (major, minor, and micro), to see if the perception of practitioners about TD evolution could be impacted. While all methods report generally growing trends of TD over time, there are different patterns. SQALE reports more periods of steady states compared to MI and SIG TD. MI is the method that reports more repayments of TD compared to the other methods. SIG TD and MI are the models that show more similarity in the way TD evolves, while SQALE and MI are less similar. The implications are that each method gives slightly a different perception about TD evolution.

1. Introduction

Technical Debt (TD) is a metaphor introduced by Ward Cunningham in 1993 [1]. Cunningham compared poor decisions and shortcuts taken during software development to economic debt. Even though these decisions can help in the short term, such as speeding-up development or the release process, there is an unavoidable cost that will have to be paid on the long term in terms of redevelopment and increased complexity for the implementation of new features, not to mention possible defects and failures.

The fundamental of this metaphor, however, was shaped in the 80s, when Lehman introduced the laws of software evolution [2]. The second law states that "*as a system evolves, its complexity increases unless work is done to maintain or reduce it*." Even though this metaphor was coined more than two decades ago (and almost 40 years passed since the definition of the software laws), the significance of this topic in the academic sphere can be observed only in the last eight years, when the number of studies has risen significantly [3].

To this day, however, there is still no clear definition of what exactly can be considered as TD. A vast amount of articles and studies were published with definitions of TD that were shaped differently [4]. Originated from the industry, the TD phenomenon has become popular first among the agile development community. Kruchten et al. pointed out that while the TD metaphor was mostly unused and not given much attention for many years, the increased interest in TD has emerged in parallel with the arrival of agile methods, increasing the interest in TD within the industry [4].

In general, the impact of TD can be quite relevant for industry and open source communities [5, 6]. Many studies found out that TD has negative financial impacts on companies [7–9]. Every hour of a developer time used on fixing poor design or figuring out how badly documented code works with other modules instead of developing new features can be considered a waste of money from the company point of view. [10]. Much like in economics, interest is present in the TD sphere as well. Cunningham describes the interest as "*every minute spent on not-quite-right code*" [1].

The goal of the paper is to compare three main techniques about TD identification that were proposed over time for source code TD identification: (i) the Maintainability Index (MI) (1994) which was one of the first (criticized) attempts to measure TD and is still in use, (ii) SIG TD models (2011) which were defined in search of proper code metrics for TD measurement, and (iii) SQALE (2011) is a framework that attempts to put into more practical terms the indication from the ISO/IEC 9126 software quality standard (recently replaced that ISO/IEC 25010 : 2011).

We take more of an *exploratory* approach with respect to previous studies comparing TD identification approaches (e.g., [11]); we do not look at how measurements collected from the approaches can be compared against common software quality metrics (useful for more *explanatory* studies), rather we compare the time series derived from different methods, looking at the results in the terms of trends and time series similarities to see how much one method can be comparable to another one. This evaluation can give us some insights about the perceptions about the amount of TD in a project that practitioners can have when applying one method over the other. For example, constantly growing trends can lead to different actions than more gradual (or even decreasing) TD evolution trends.

The paper is structured as follows. In Section 2, we provide the background about TD: definitions over time, methods used to compute TD, and related studies attempting to compare TD methods. In Section 3, we present the three methods that are compared in the paper: MI, SIG TD, and SQALE. In Section 4, we describe the experimental evaluation and the results: methods, projects used, differences emerging from the application of the three methods. Section 6 concludes the paper.

2. Background

2.1. TD Definitions over Time. One of the widely spread definitions of TD is from McConnell in 2008: "A design or construction approach that's expedient in the short term but that creates a technical context in which the same work will cost more to do later than *it would cost to do now (including increased cost over time)*" [12]. However, even if his taxonomy of TD is often used in scientific papers or tech blogs, there are still some points of conflicts within the community. The most discussed point is *unintentional debt* described by McConnell as "a design approach that just turns out to be error-prone or a junior programmer just writes bad code." This was criticized among others by Martin [13] and backed up by Norton [14].

Martin describes TD as short-term engineering tradeoffs or suboptimal designs which are appropriate in schedule-driven projects, as long as the benefit of the indebtedness, for example, the early release, drives the business to repay the debt in the future [13]. Even Cunningham later noted that the original definition of TD was meant to contain design decisions and not the quality of the code, noting that "the ability to pay back debt [...] depends upon you writing code that is clean enough to be able to refactor as you come to understand your problem" [15]. Contrary to Martin's and Norton's beliefs, many members of the development community think that bad and legacy code and old technologies are part of TD too. Fowler [16] and Atwood [17] both mention the terms "*quick, dirty*, and *messy*" when referring to TD definitions from Cunningham and Martin.

In the research context, there were many interpretations of TD as well. Guo et al. presented TD as "*incomplete, immature, or inadequate artifact in the software development lifecycle*" [18]. Theodoropoulos proposed a new, broader definition of TD: "technical debt is any gap within the technology infrastructure or its implementation which has a material impact on the required level of quality" [19]. Gat proposed an even more extensive definition of TD: "*quality issues in the code other than function/feature completeness*" divided into intrinsic and extrinsic quality issues [20]. Sterling also agrees that functionality or feature completeness has to be excluded from the TD definition because even infested with TD, the application should meet a minimum standard of satisfaction for its users [21].

Even though there are many different interpretations of the TD metaphor in the academic publications, there seems to be a consensus that even a bad code should be considered as TD. Curtis implies that TD should be considered as everything that is "*must-fix*" from the point of view of the software company. The TD metaphor has its limits, however. It was noted that, since it was coined more than two decades ago, it is not sufficient enough when modern development techniques, such as agile or extreme programming (XP), are used [22]. In these techniques, the product is purposely delivered unfinished, but the TD incurred by this approach is controlled and often repaid in the next iteration.

One of the most important shortcomings of TD definitions is the fact that there is yet to be a unified measurement unit [23]. It is generally complex to quantify most of the forms of TD. Due to complexity of the TD concept, attempts to classify TD and the creation of holistic frameworks emerged over time. Tom et al. [9] divide technical debt into several categories: *code debt*, *design and architectural debt*, *environment debt (connected to the hardware software ecosystem)*, *knowledge distribution and documentation debt*, and *testing debt*. TD identification techniques proposed over time generally look at either one of the aspect of combination of several aspects into one framework for measurement. As evident from the categories, different metrics need to be considered for each of the categories.

The most common way to quantify source code TD is to track code TD, such as code smells [24] or similar constructions that do not follow conventions and best practices of a given language. If we exclude code-level debt from TD, the quantification of TD becomes much harder, for example, at the architectural level [25].

2.2. Analogy between TD and Financial Debt. At the basis of the software TD popularity, there is the analogy between TD and financial debt [9].

2.2.1. Monetary Cost. TD has generally a negative financial impact on companies [7–9]. Costs of fixing poor design or badly documented code can be considered a waste of money from a company point of view. These costs can be also indirect, such as the negative effect on the morale of the team [10].

2.2.2. Interest and Principal. As in the financial context, the concept of interest is also present within TD definition. In the context of software development, Cunningham describes the interest as "every minute spent on not-quite-right code" [1]. Camden extends this definition by adding the needs of repayments of the established debt (e.g., paying the principal, that is, the amount borrowed) [26]. In the software context, additional effort needs to be made to work around the TD debt within a project. This extra effort is the interest that has to be paid.

Edith Tom points out that one of the forms of the interest costs is also a decrease in the developers' productivity, and it can lead to accumulation of more debt in the form of slower development speed [9]. Some authors, such as Ries, report that not all debts are equal. The interest rates are different based on whether the code will be in use.

2.2.3. Repayment and Withdrawal. Repayment of TD can be interpreted as refactoring the previous source code solutions. Short-term debt can (and should) be repaid easily, preferably in an early next release. However, it is much easier to incur TD unintentionally rather than consciously and "*much harder to track and manage after it has been incurred*" [12]. Similar to the financial context, McConnell suggests a theory of credit ratings, which can be associated with the withdrawal of TD. He describes a team's credit ranking as "*a team's ability to pay off technical debt after it has been incurred*" [12]. This ability might take into account various factors, most notably if the development velocity started to decrease as a consequence of the TD evolution. Based on the variety of the repayments, different forms of TD can be defined. One of the examples is about the visibility of TD (e.g., because of the size of the debt compared to the whole system sizing) and therefore it can require extra work to recognize the needs of repayments [12].

2.2.4. Bankruptcy. Stockpiling of TD can eventually lead to bankruptcy. Contrary to the financial bankruptcy, in software development, bankruptcy means that any further development is not possible anymore and a complete code rewriting is necessary. The point when it is necessary to file for bankruptcy can vary in different projects and by different developers. As it is common in the TD phenomenon, bankruptcy is defined differently among the experts. An application can be considered bankrupted when "the cost of improving the existing code in an application may indeed be greater than the cost of rewriting it" [27]. On the contrary, Hilton describes bankruptcy as the moment when "you will either suspend all feature development to pay down all of your debt at once, or you will have to rewrite the entire application" [28]. Tom et al. describes the bankruptcy as the point in SW development "when tactical, incremental, and inadvertent forms of technical debt (which do not have a view to the long term) are left unmanaged without any controls in place" [9]. Even though the exact point of bankruptcy is defined differently, all of these theories have something in common. The worst consequence of the bankruptcy is the necessity to rewrite whole software, which essentially stops any further development and causes the loss of even more time and money.

2.2.5. Leverage. Leverage is one of the reasons of deliberate and conscious TD inclusion into the codebase. The time saved on development can lead to earlier releases, which can be crucial, for example, for start-ups. A certain amount of TD does not have to be necessarily considered a bad thing when managed well and constrained to specific limits. McConnell notes that it is necessary to manage TD at a reasonable level since it can slow down development velocity in the future. Because of this reason, companies have to invest time in managing and paying down TD rather than accumulating TD [12]. A similar idea was mentioned by Fowler: "you need to deliver before you reach the design payoff line to give you any chance of making a gain on your debt... even below the line, you have to trade-off the value you get from early delivery against the interest payments and principal pay-down that you'll incur" [16].

3. Three Metrics for Measuring TD

Many methods emerged for TD identification [29–33]. In this paper, we focus on three alternative methods that were used over time for TD identification: (i) the Maintainability Index (MI), (ii) SIG TD models, and (iii) SQALE analysis. There are multiple reasons for the selection of these methods. On one side, these are popular methods that were proposed over time, but no comparison was performed yet. On the other side, these methods (or adaptations) are all metrics-based so they assume similar effort in the collection of the necessary data. Furthermore, they are available in code editors and can currently be used by practitioners, so a comparison can be useful to understand the differences, being the Maintainability Index one of the first models to be proposed.

3.1. Maintainability Index (MI). At the International Conference on Software maintenance, in 1992, Oman and Hagemeister introduced an article, which collected 60 metrics for gauging software maintainability [34]. Several of these metrics were hard to compute, mainly because most of them required either historical or subjective data. Their goal was to come up with the minimal set of easily computed requirements, based on which it would be possible to predict software maintainability. In the refined and published article, in 1994, they introduced the Maintainability Index (MI) [35].

To find a simple, applicable model which is generic enough for a wide range of software, a series of 50 statistical regression tests were used. Out of 3 models which were tested on additional test suite (along with another questionnaire), a four-metric polynomial based on Halstead Volume, McCabe Cyclomatic complexity, Lines of Code, and Lines of Comments was chosen as a Maintainability Index. The original formula of MI was defined as follows:

$$\begin{aligned}\mathrm{MI} = 171 - 5.2\ln(\mathrm{HV}) - 0.23(\mathrm{CC}) - 16.2\ln(\mathrm{LoC}) \\ + 0.99(\mathrm{CMT}),\end{aligned} \tag{1}$$

where HV is an average Halstead Volume per module, CC is an average Cyclomatic Complexity per module, LoC is average lines of code per module, and CMT is average lines of comments per module.

Despite the popularity of the Maintainability Index, it is still viewed as a controversial metric. It is criticized for various reasons, such as a not clear explanation of the formula, the usage of averages per file, using the same formula from 1994, and possibly some ambiguous connection from the results to specific source code metrics [36].

However, a derivative formula of MI is still used in some popular code editors (e.g., Microsoft Visual Studio), so it is relatively easy to be adopted by practitioners to detect TD in their projects. In our analysis, we use it as one of the baseline metrics for the comparison, to see how such metric compares with more recent metrics.

In the experimental evaluation, we used the derivative formula adopted in Microsoft Visual Studio, which gives the MI a range (0, 100), compared with the original 171 to an unbounded negative number, of difficult interpretability:

$$\mathrm{MI} = \max\left[0, 100\,\frac{171 - 5.2\ln(\mathrm{HV}) - 0.23(\mathrm{CC}) - 16.2\ln(\mathrm{LoC})}{171}\right]. \tag{2}$$

3.2. SIG TD Models. The Software Improvement Group (SIG) defined, in 2011, a model which quantifies TD based on an *estimation of repair effort* and *estimation of maintenance effort*, which provides a clear picture about the cost of repair, its benefits, and the expected payback period [33]. The model is based on the SIG maintainability model and computes the technical debt and interest based on the empirically defined values and metrics.

The SIG maintainability model was created based on a set of minimal requirements based on the limitations of other metrics such as the Maintainability Index [37]. The model maps system characteristics (taken from ISO 9126) onto source code properties. Each source code property can be evaluated by easily obtainable, language-independent metrics:

Volume: lines of code or man years via function points.

Complexity: McCabe's cyclomatic complexity per unit. A unit is the smallest independently executable and testable subset of code for a given language.

Duplication: percentage of repeated blocks of code (over six lines long).

Unit size: lines of code per unit.

Unit testing: unit test coverage or number of assert statements.

Each of these metrics is graded on a 5 point scale based on their values [37]. To get the overall grade of system characteristics, we have to compute the rounded average of source code properties mapped to the particular system characteristic. For example, to get the grade of analysability, we need to get the average value out of volume, duplication, unit size, and unit testing (see Table 1).

This model uses the grading scale of the SIG maintainability model. It proposes the overall grade of the project as an average of the system characteristics. The unit of maintainability is a "*star*," so if the average value of system characteristics is for example 5, they refer to it as a 5-star project.

Quantifying TD of the project is performed in several steps and requires a calculation of three different variables: rebuild value (RV), rework fraction (RF), and repair effort (RE).

Rebuild value is defined as an estimate of the effort (in man-months) that needs to be spent to rebuild a system using particular technology. To calculate this value, the following formula is used:

$$\mathrm{RV} = \mathrm{SS} \times \mathrm{TF}, \tag{3}$$

where SS is System Size in Lines of Code and TF is a Technology Factor, which is a constant that represents language productivity factor [38, 39].

Rework Fraction is defined as *an estimate of % of LoC to be changed in order to improve the quality by one level*. The values of the RF in between two quality levels are empirically defined [33].

Finally, the *Repair Effort* is calculated by the multiplication of the Rework Fraction and Repair Effort. It is possible to multiply it by Refactoring Adjustments (RA) metric, which shows external, context-specific aspects of the project, which represent a *discount* in the overall technical debt of the project:

$$\mathrm{RE} = \mathrm{RF} \times \mathrm{RV} \times \mathrm{RA}. \tag{4}$$

3.3. SQALE. Software QuALity Enhancement (SQALE) focuses on the operationalization of the ISO 9126 Software Quality standard, by several code metrics that are attached to the taxonomy defined in ISO 9126 [31]. Mimicking the ISO 9126 standard, SQALE has a first level defining the characteristics (e.g., testability), further subcharacteristics (e.g., unit testing testability), and further source code level requirements. An example characteristic can be testability, that is how much testable is the system, that can be divided into integration testing testability and unit testing testability subcharacteristic. Source code requirements could be, for example, a coupling between objects (CBO) metric <7 and the number of parameters in a module call (NOP) <6. Figure 1 gives an example of the association between quality characteristics and subcharacteristics and code metrics. At the end, the source code requirements are then mapped to

Table 1: Mapping matrix of the SIG model, example of analysability (adapted from Heitlager et al. [37]).

	Source code properties				
	Volume	Complexity per unit	Duplication	Unit size	Unit testing
Maintainability	x		x	x	X
Analysability					
Changeability		x	x		
Stability					x
Testability		x		x	x

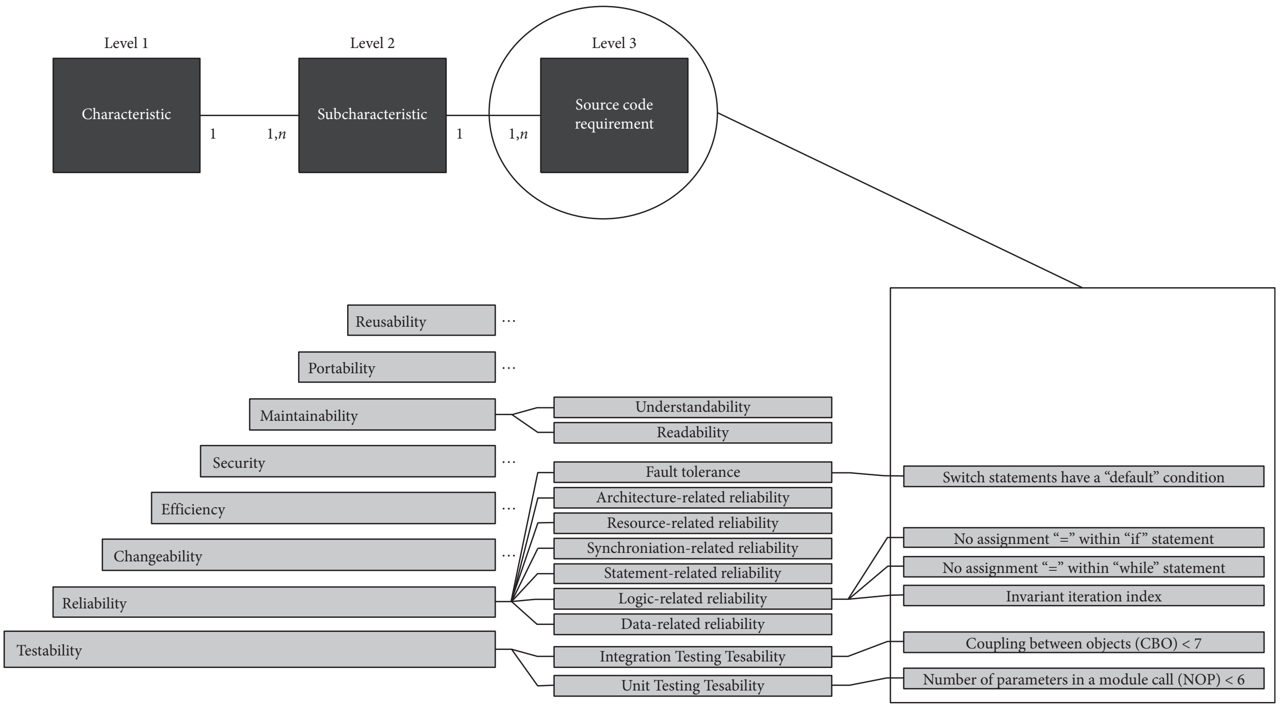

Level 1
Level 2
Level 3
Characteristic
Subcharacteristic
Source code requirement
1
1,n
1
1,n
Reusability
Portability
Maintainability
Security
Efficiency
Changeability
Reliability
Testability
Understandability
Readability
Fault tolerance
Architecture-related reliability
Resource-related reliability
Synchroniation-related reliability
Statement-related reliability
Logic-related reliability
Data-related reliability
Integration Testing Tesability
Unit Testing Tesability
Switch statements have a "default" condition
No assignment "=" within "if" statement
No assignment "=" within "while" statement
Invariant iteration index
Coupling between objects (CBO) < 7
Number of parameters in a module call (NOP) < 6

Figure 1: SQALE. Example of metrics related to software quality characteristics.

remediation indexes that translate in the time/effort required to fix the issues. The aggregation of all the characteristics gives the overall TD.

For calculation of TD, SQALE can be used by computing the so-called Remediation Cost (RC), which represents the cost to fix the violations to the rules that have been defined for each category [40]:

$$\mathrm{RC} = \frac{\sum_{\mathrm{rule}} \mathrm{effortToFix}\left(\mathrm{violations}_{\mathrm{rule}}\right)}{8\ [\mathrm{hr/day}]}. \tag{5}$$

For SQALE, we adopted the SonarQube implementation: a default set of rules was used, which is claimed to be *the best-practice, minimum set of rules* to assess the technical debt.

4. Experimental Evaluation

For the definition of the experimental evaluation, we defined the following goal: to analyze technical debt evaluation techniques (MI, SQALE, and SIG TD) for the purpose of comparing their similarity with respect to the trends and evolution of the measurements from the point of view of practitioners aiming at measuring TD.

The goal was refined into three main research questions (RQs):

RQ1: are the trends of the measurements provided by the three methods comparable? *Metric: Pearson correlation between trends in time series.*

RQ2: how are trends of TD comparable across different release types? *Metric: comparison of trends by release type.*

RQ3: how much can one method be used to forecast another one? *Metric: Granger causality between time series.*

4.1. Datasets. The tested packages were randomly selected from the list of 5000 most popular Python libraries (https://hugovk.github.io/top-pypi-packages/, manuscript submitted to ACM). However, we defined some criteria due to the metrics that needed to be computed and the implementation steps necessary for the analysis:

Repository on GitHub to be mined

At least 5 releases published in the repository in Python 3

Traditional major.minor.micro release notation

Packaged via setup tool library

Possibility to run unit tests via setup.py command

While almost all selected packages had their repository on GitHub, some, especially less popular packages had too few release versions available. Furthermore, if the selected library was written solely in Python 2.x version, it was not considered, as the analysis was implemented in Python 3.6 and the incompatibilities could affect the results. Similarly, the vast majority of the packages used standard Major.Minor.Micro release notation. However, according to Python Enhancement Proposal (PEP) 481, date-based release segments are also permitted [41]. For simplicity and unified versioning, such libraries were omitted from the list of possible candidate projects. Overall, 20 projects were selected: *P1. Blinker*, *P2. Colorama*, *P3. CSVKit*, *P4. Dateparser*, *P5. Decorator*, *P6. Deprecation*, *P7. Grip*, *P8. ISODate*, *P9. JmesPath*, *P10. KeyRing*, *P11. McCabe*, *P12 Pyasn*, *P13. PyFlakes*, *P14.PythonISO3166*, *P15. TinyCSS2*, *P16. Yamllint*, *P17.Yapf*, *P18. Fakeredis*, *P19. influxdb-python*, and *P20. pylint* (the full list can also be found in Table 2, and in [42]). These are all Python libraries that are used to provide functionalities to other applications, such as *csvkit*, useful to provide support for management of csv formats.

4.2. Rationale and Methods. To compare the three methods, we look at the time series of all the measures collected by all the three methods. For each of the projects, we implemented the method for analysis, we aggregated all measures by release dates, and built time series of each metric that we used as baseline for running the analyses. As each of the TD identification methods has different scales that can be difficult to directly compare, we look at the trends and time series behaviours resulting from the application of all the methods.

We define a time series as T, consisting of data points of TD at each release time $R = \{r_1, r_2, \ldots, r_n\}$, as $T = \{t_{r1}, t_{r2}, \ldots, t_{rn}\}$. An example of the time series for the three methods considered can be seen in Figure 2, where we plotted releases and TD measurements for the xmllint project. As can be seen, the MI measure is an inverse of the other measures, as is giving an indication of the maintainability of the project (the lower the worse), while the other methods give indication of TD accumulating (the higher the worse). For the other parts of the analysis, to compare the time series, we reversed the MI index, to make it comparable to the other approaches (e.g., for the correlation of the trends). For RQ1 trends of measurements, we compute TD's Δ measurements between two releases for each of the projects.

For release r_i, ΔTD is defined as follows:

$$\Delta TD_{ri} = \frac{(t_{r1-1} - t_{r1})}{(t_{r1-1} + t_{r1})/2}. \tag{6}$$

We then compute the Pearson correlations between all the points of each of the compared methods. Results of the ΔTD_{ri} for each of the time series are also shown in an aggregated form in boxplots. Given all the changes in trends of TD measurements, the comparison of boxplots can showcase the differences in trends between the three TD identification methods.

For RQ2, we delve into the trends for the different types of project releases: major (e.g., 0.7.3, 1.0.0, 2.0.0), minor (e.g., 0.7.3, 0.8.0, 0.9.0), and micro (e.g., 0.9.0, 0.9.1, 0.9.2) releases, looking at whether considering different type of releases can have an impact on the results given by all the methods. To answer this research questions, we look at TD's $\Delta\uparrow$ as increasing trends, $\Delta\downarrow$ as decreasing trends, and $\Delta 0$ in periods between releases in which TD did not change, where ΔTD_{ri} is categorized in one of the categories:

$$\Delta TD = \begin{cases} \Delta TD\uparrow, & \text{if } \Delta TD_{ri} > 0, \\ \Delta TD\downarrow, & \text{if } \Delta TD_{ri} < 0, \\ \Delta TD-, & \text{otherwise.} \end{cases} \tag{7}$$

For RQ3, we look at how much one of the three methods can be used to forecast the results from another method.

We take into account time series of the measurements from the three methods (MI, SIG TD, and SQALE), and we compute Granger causality between methods in pairs.

Granger causality test, first proposed in 1969 by Clive Granger, is a statistical hypothesis test which is used to determine whether a time series can be used to predict other time series values [43]. More precisely, we can report that $T1$ "*Granger causes*" $T2$, if the lags of $T1$ (i.e., $T1_{t-1}, T1_{t-2}, T1_{t-3}, \ldots$) can provide predictive capability over $T2$ beyond what allowed by considering the own lags of $T2$.

The null hypothesis is that T2 does not Granger-cause the time series of T1. We adopted the *standard SSR-based F test*. If the probability value is less than *0.05*, it can be concluded that T2 Granger-causes T1.

4.3. Results

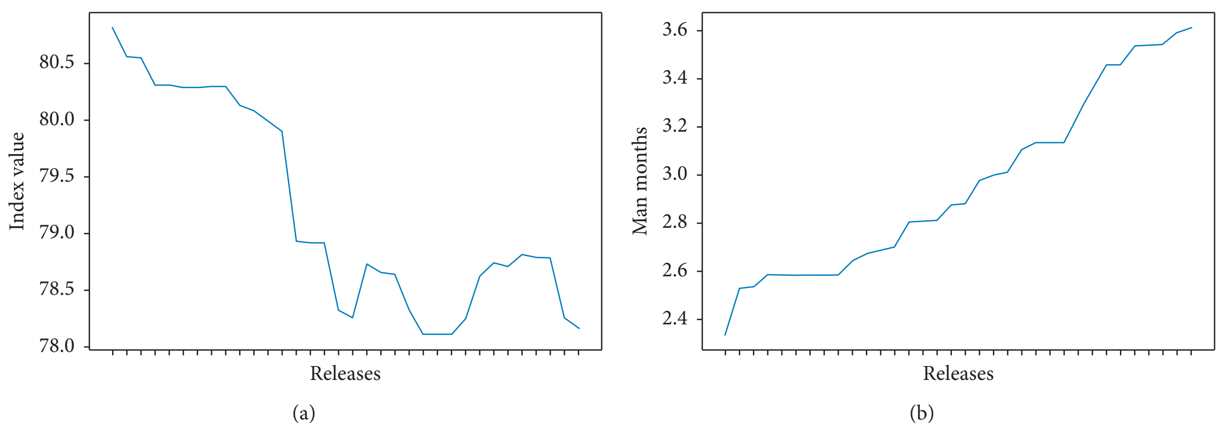
4.3.1. RQ1: Are the Trends of the Measurements Provided by the Three Methods Comparable? To compare TD over time between different techniques, the Pearson correlation was used. The bivariate Pearson correlation measures the strength and direction (−1.0, +1.0) of linear relationships between pairs of variables.

Figure 3 reports the boxplots of the correlation between the trends for each release (ΔTD_{ri}). Each datapoint in the boxplot constitutes the correlation for one project. The three boxplots propose the comparison SQALE-SIG (median: 0.74), SQALE-MI (median: 0.57), and SIG-MI (median: 0.67).

The figure shows that SQALE and MI are the least comparable methods, having the highest number of negative

Table 2: Python software projects analyzed (LoCS = Line of Code Statements and TC = Test Coverage refer to the latest release).

Project	URL	LoCS	TC (%)	Releases
P1. blinker	https://pypi.org/project/blinker	874	28	0.8–1.4
P2. colorama	https://pypi.org/project/colorama	1 471	59	0.3.3–0.4.1
P3. csvkit	https://pypi.org/project/csvkit	2 906	87	0.8.0–1.0.4
P4. dateparser	https://pypi.org/project/dateparser	3 339	81	0.3.0–0.6.0
P5. decorator	https://pypi.org/project/decorator	2 758	88	4.2.0–4.3.2
P6. deprecation	https://pypi.org/project/deprecation	1 083	100	1.0.1–2.0.6
P7. grip	https://pypi.org/project/grip	1 763	30	3.0.0–4.5.2
P8. ISODate	https://pypi.org/project/isodate	2 794	94	0.4.9–0.6.0
P9. jmespath	https://pypi.org/project/jmespath	1 714	72	0.0.1–0.9.4
P10. keyring	https://pypi.org/project/keyring	2 242	56	1.0–19.0.1
P11. McCabe	https://pypi.org/project/mccabe	641	79	0.1–0.6.1
P12 pyasn1	https://pypi.org/project/pyasn1	10 014	82	0.2.3–0.4.5
P13. pyflakes	https://pypi.org/project/pyflakes	8 732	93	0.7.3–2.1.1
P14. pythonISO3166	https://pypi.org/project/iso3166	380	56	0.1–1.0
P15. tinyCSS2	https://pypi.org/project/tinycss2	2 796	99	0.1–1.0.2
P16. yamllint	https://pypi.org/project/yamllint	3 541	98	0.6.0–1.15.0
P17. yapf	https://pypi.org/project/yapf	5 357	96	0.1.4–0.9.0
P18. fakeredis	https://pypi.org/project/fakeredis	8 395	97	0.4.2–1.2.1
P19. influxdb-python	https://pypi.org/project/influxdb	9 217	85	0.1.13–5.2.3
P20. pylint	https://pypi.org/project/pylint	28 669	90	1.7.0–2.4.4



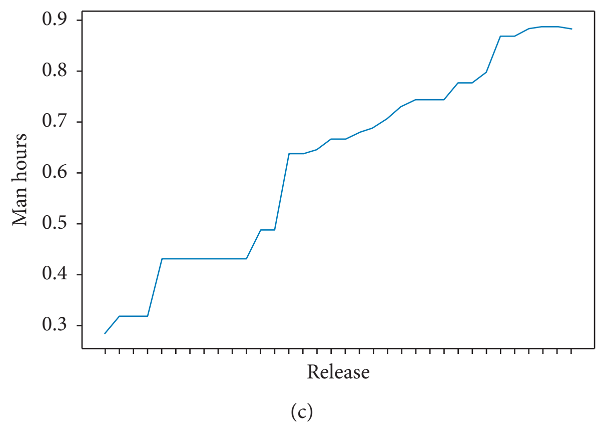


Figure 2: Example of plotted TD time series (MI, SIG, and SQALE) for the yamllint python library. (a) MI, (b) SIG TD, and (c) SQALE.

correlation and much wider variability than the other models compared. SQALE and SIG and SIG and MI showed similar distributions of the correlations, slightly in favor of SIG and MI, which have less negative correlations and also lower variance. To look if such differences are statistically significant, we run Wilcoxon Signed-Rank Tests and paired difference tests to evaluate the mean rank differences for the correlations. The difference is not statistically significant for

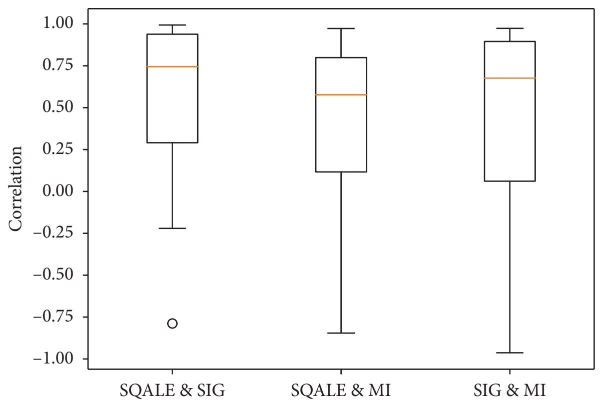


Figure 3: ΔTD_{ri} correlation between different methods.

SQALE-MI vs. SIG-MI (p value 0.2801, $p \geq 0.05$, and two-tailed) and not significant for SQALE-SIG vs. SQALE-MI (p value 0.496, $p \geq 0.05$, and two-tailed).

When we look at the trends on comparisons between every two following releases (Table 3), the trend is similar for SIG and MI (as previous correlations discussed), with a slight difference in the falling trend. This seems to indicate that, according to MI, TD tends to be repaid more often than on SIG. In SQALE, however, we can observe that TD was more stable across different releases (see Table 3). We can further see the variability of trends of TD identification for all projects in Figure 4. For each project, we plot the boxplots of trends of variations of TD between every two consecutive releases. For example, we can see that *P1 blinker* has only growing trends of 0 to +35% (as an outlier), while *P2 colorama* has not only positive trends up to +7.5% but also periods of negative trends down to −12.5%, in which TD decreases. Looking at *P1 blinker* project, we can see that SQALE has much more variation in trends compared to the other two methods with MI having much lower changes in trends. Looking at the comparison, we can see that some projects have some decreasing trends, though these changes are rather limited, confirming that TD is generally reported as increasing by all methods. Each project is also rather specific, such as isodate, that reports a large variation in terms of MI trends, while generally MI measurements are the trends with less variations in other projects.

RQ1 findings: considering the correlation between trends of TD changes between releases, SIG TD and MI are the models which show more comparability in terms of correlation of the trends of TD changes. SQALE and SIG TD show less similarities. Generally, SQALE and MI are the models that show lower correlation in trends in the 20 projects considered. SQALE is also the model that shows more stable periods of debt compared to the other methods.

4.3.2. RQ2: How Are Trends of TD Comparable across Different Release Types? This RQ is similar to RQ1, but in RQ2, we look at the comparison based on the release types, that is, if major, minor, and microreleases matter for the differences in TD identification among the three methods.

Table 3: TD trends on all releases.

	$\Delta TD\uparrow$ (%)	$\Delta TD\downarrow$ (%)	$\Delta TD-$ (%)
SQALE	34.52	6.07	59.41
SIG TD	74.68	11.72	13.6
MI	57.74	20.08	22.18

Comparisons solely between *major* releases have brought interesting results (Table 4), similar to the results on all releases. Throughout all comparisons, most of *major* releases caused TD to rise for each analysis. SQALE had again the highest number of still trends and the most TD repayments (falling trend) were recorded with MI.

The rising of TD is stronger at *minor* release level for both SIG TD and SQALE, as each of the methods encountered the rise of rising trend compared to *major* releases (see Table 5). SQALE showed a decrease in growing trends. As in the previous cases, SQALE recorded more periods of steady TD, and MI the most repayments of TD (to a much larger extent than SIG TD and SQALE).

The last comparison was carried out on *micro*releases. The same trends were observed also at this level: vast majority of releases inducted more TD on SIG and MI, while considering SQALE the majority of releases did not change TD (see Table 6). Again, MI is the method that reports more TD repayment (21.99%).

RQ2 findings: considering *major, minor,* and *micro*-releases, MI and SIG TD show mostly the majority of growing trends. SQALE shows the most of TD steady states, while MI shows much larger TD repayment periods compared to the other methods. These patterns seem to be consistent across *major, minor,* and *micro* releases. They show that the selection of a specific method can have an impact on the perception of the presence of technical debt in a project.

4.3.3. RQ3: How Much Can One Method Be Used to Forecast Another One? A time series X can be said to Granger-cause another time series Y if the probability of correctly forecasting Y_{t+1}, with $t = 1, \ldots, T$, increases by including information about X_t in addition to the information included in Y alone. In the case of the three methods reviewed, this means that the measurements from one method (e.g., SQALE) can be used together with the measurements of another method (e.g., MI), to provide better prediction of future values of the complemented method (e.g., MI). Aggregated results for all projects about Granger causality tests can give us the indication about how much a time series results from a TD identification technique can help in forecasting time series from the other method.

To note that Granger causality, differently from correlation, is generally an asymmetric property, that is, the fact that, in one, the projects' time series SQALE Granger-causes SIG TD model does not imply that SIG TD model Granger-causes SQALE results. For this reason, we provide all the combinations of results with counters for how many times the results were positive according to the F-test taking into account all the 20 projects analyzed (Table 7).

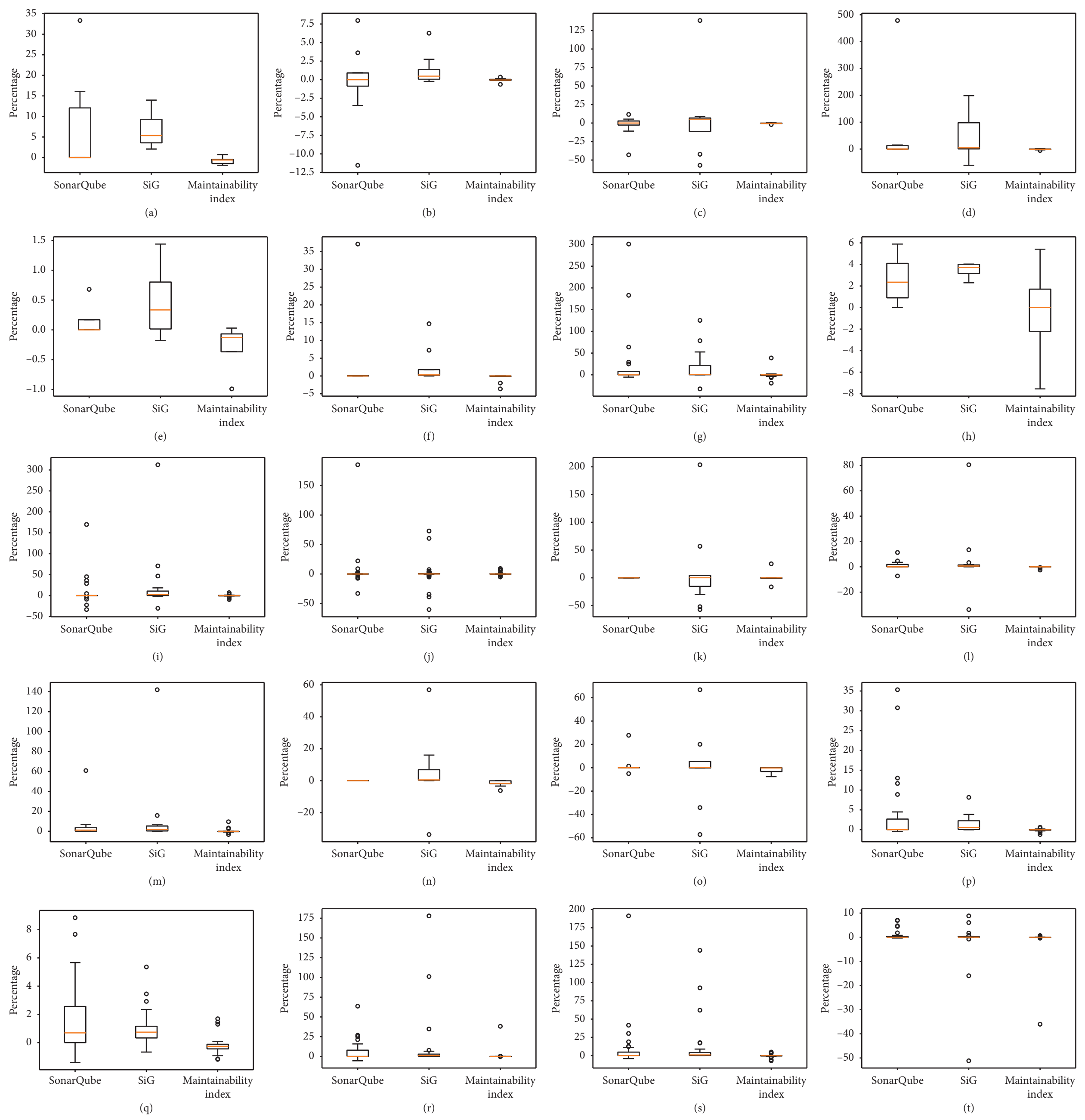


Figure 4: Variability of trends for each of the methods (SQALE, SIG TD, and MI): y-axis represents % of changes in the measurements between different releases (note different scales). (a) (P1) *blinker*, (b) (P2) *colorama*, (c) (P3) *csvkit*, (d) (P4) *dateparser*, (e) (P5) *decorator*, (f) (P6) *deprecation*, (g) (P7) *grip*, (h) (P8) *isodate*, (i) (P9) *jmespath*, (j) (P10) *keyring*, (k) (P11) *mccabe*, (l) (P12) *pyasn*, (m) (P13) *pyflakes*, (n) (P14) *pythoniso3166*, (o) (P15) *tinycss2*, (p) (P16) *yamllint*, (q) (P17) *yapf*, (r) (P18) *fakeredis*, (s) (P19) *influxdb*, and (t) (P20) *pylint*.

Table 4: TD trends on major releases.

	$\Delta TD\uparrow$ (%)	$\Delta TD\downarrow$ (%)	$\Delta TD-$ (%)
SQALE	70.27	10.81	18.92
SIG TD	75.68	24.32	0
MI	56.76	43.24	0

Table 5: TD trends on minor releases.

	$\Delta TD\uparrow$ (%)	$\Delta TD\downarrow$ (%)	$\Delta TD-$ (%)
SQALE	46.03	8.47	45.5
SIG TD	87.83	6.35	5.82
MI	67.72	20.11	12.17

Table 6: TD trends on microreleases.

	$\Delta TD\uparrow$ (%)	$\Delta TD\downarrow$ (%)	$\Delta TD-$ (%)
SQALE	40.15	7.06	52.79
SIG TD	74.72	13.38	11.90
MI	55.02	19.70	25.28

Table 7: Aggregated results of Granger causality tests (percentage of F-test results).

	True (%)	False (%)	None (%)
SQALE-MI	30	60	10
SQALE-SIG	25	65	10
SIG-SQALE	25	65	10
MI-SIG	15	85	—
MI-SQALE	10	80	10
SIG-MI	10	90	—

It is noteworthy to say that, in case of two tested libraries, the linear trend of TD in SQALE caused the tests to end with an error because Granger causality test does not capture instantaneous and nonlinear causal relationships. In general, SQALE-MI and SQALE-SIG TD had F-statistic significant in around 1/3 of the projects (30%), while in the majority of the other cases Granger causality was negative. These results could indicate that, in some of the results, considering the lagged values of SQALE time series results, we can get better prediction of values in MI and SIG TD time series.

RQ3 findings: results from Granger causality show that there is a limited relation between the different methods compared, as indication that TD identification measurements are rather independent. Only SQALE time series Granger-causes both MI and SIG TD in 1/3 of the projects, with SQALE-SIG, and SIG-SQALE also near in terms of positive Granger causality. For other methods, there is mostly no Granger causality.

4.3.4. Replicability. The analysis was implemented and run using Python version 3.6.3. A derivative from *Radon* library was used (https://radon.readthedocs.io/en/latest/index.html). Radon is a Python package which computes various source code metrics, such as McCabe's Cyclomatic complexity, Halstead metric, raw metrics (lines of code and lines of comments), and MI. The *sonar-python* plugin (https://docs.sonarqube.org/display/PLUG/SonarPython) was used for the SQALE analysis. The stats model library was used for computing the Granger causality tests. Scripts can be run to run, aggregate, and plot all the results in a semiautomated way.

A replication package contains all the metrics collected for all the versions of the analyzed projects, together with the raw analyses, diagrams, and source code used for the analysis [44].

4.4. Threats to Validity

4.4.1. External Validity. Threats to external validity are related to the generalizability of our study [45]. The empirical evaluation was conducted on 20 Python projects extracted from a large sample. The results on other set of projects could be different, also peculiarities of Python projects might play a role. The projects that were included are all small Python libraries useful for functionalities in other Python applications. Even though these are small libraries, we followed the evolution over time and all the commits performed to reconstruct larger item set for data analysis.

4.4.2. Internal Validity. Threats to internal validity are related to experimental errors and biases [45]. The data collection process has been carried out to make it as much replicable as possible: by running some scripts is possible to start over the collection process, calculating the TD measurement, and then aggregating the results in the visual and textual form. One issue is that calculation of coverage of projects (for coverage measures in the SIG TD model) implies that projects have to be built and tested. This introduces a phase that is semiautomated, as projects that could not be built either need to be fixed or cannot be included in the sample set of projects. Another internal validity threat is about the implementation of the method. For MI, we used the implementation from the Radon library, and for SQALE, we reutilized SonarQube implementation through the *sonar-python* plug-in. The SIG TD models were reimplemented based on the information available in the papers published about the models (e.g., [33]). Either in our or the adopted implementation, there might be some inconsistencies or issues in the calculations of the metrics, being difficult to have some tests for TD ground truth [46].

4.4.3. Construct Validity. Threats to construct validity relate to how much the measures used represent what researchers aim to investigate [45, 47]. In our case, we considered black-box analyses based on trends of evolution of the measurements provided by the three TD identification techniques. Our definition of trends was connected to software releases (*major, minor,* and *micro*), which we considered appropriate due to the research questions that were set. The current study was more exploratory in looking for emerging patterns, while further studies can look into more explanatory aspects.

5. Related Works

In the literature, TD has been investigated from various perspectives [9, 48] and number of different approaches has been proposed for TD identification and management [49].

However, there are not many studies that compare alternative TD identification methods. One reason could be the complexity/time required to implement the methods and the second reason about the comparability of the metrics defined. Furthermore, Izurieta et al. [46] note that it can be difficult to compare alternative TD measurements methods due to missing ground truth and the uncertainties of the measurement process. One of the earliest studies to compare metrics for TD identification was the study by Zazworka et al. [11], comparing four alternative methods across different versions of Apache Hadoop: (a) modularity violations, (b) design patterns grime build-up, (c) source code smells, and (d) static code analysis. The focus was on comparing how such methods behave at the class level. The findings were that the TD identification techniques indicate different classes as part of the problems, with not many overlaps between the methods.

Griffith et al. [50] compared ten releases of ten open source systems with three methods of TD identification ((i) SonarQube TD plug-in, (ii) a method based on TD identification using a cost model based on detected violations, and (iii) and one method defining design disharmonies to derive issues in quality). These methods were compared against software quality models. Authors found that only one method had a strong correlation to the quality attributes of reusability and understandability.

Mayr et al. [40] proposed a benchmarking model for TD calculation. The purpose of the model is to allow comparison of a project's TD values with baseline values based on a set of reference projects. Within their work, the authors also propose a general classification scheme for the technical debt computation approaches. Additionally, authors compare 3 different TD computation approaches: CAST, SQALE, and SIG quality model. Based on the comparison, the authors derive requirements for the benchmarking model. The comparison focuses on higher-level qualitative attributes, such as data sources, target level, and productivity factors rather than on quantitative comparison on given set of projects.

Oppedijk [51] compared the statistical correlation of the results from the application of both Maintainability Index and the SIG Maintainability Model to 73 software projects: 52 proprietary software systems and 21 open source systems. 19 systems were written in the C programming language, 11 in C++, and 43 in Java. Overall, the two models were found to have a moderate statistically significant positive correlation in case of the C programming language (0.494 or 0.476 depending on the MI model applied), as well as for Java (0.459 and 0.500), while for C++ results were not significant, mainly due to the low sample sizes (0.365 and 0.423). Overall, expectations were for a higher correlation level between the two models. Another interesting finding is about the collinearity of different components of the two models, showing that some components have too much collinearity (namely, changeability vs. analysability and changeability vs. testability for the SIG Maintainability Model, average Halstead Volume, average extended cyclomatic complexity, and average unit size for MI). This suggests that some models can be improved by removing some of the components, in case of the computation of a global maintenance index.

Griffith et al. [52] introduced a conceptual model for a discrete-event simulation of the Scrum agile process which includes both defect and TD creation. The simulation is used to study the integration of multiple TD management strategies in agile development processes. To evaluate the differences between the TD management strategies, authors use five metrics: cost of completed items (CC), count of work items completed (WC), cost of effective technical debt (ETD), cost of potential technical debt (PTD), and cost of total technical debt (CTD). CC, ETD, PTD, and TD are expressed in terms of source lines of code (SLOC).

Li et al. [53] analyzed 13 open source projects with multiple releases for each of them focusing on evaluation of architectural technical debt (ATD). The ATD is measured as the average number of modified components per commit (ANMCC). In the case study, the authors investigated the correlation between the ANMCC and source code-based modularity metrics, such as Index of Package Changing Impact (IPCI), Index of Inter-Package Extending (IIPE), or Index of Package Goal Focus (IPGF). The study showed that IPCI and IPGF have strong negative correlation with ANMCC and therefore could be used as an ATD indicator. Although multiple metrics were compared, the authors did not focus directly on the comparison of existing TD measures, and they also focus solely on the ATD instead of general TD.

Kosti et al. [54] distinguished between two categories of TD approaches: methods that monetize the amount of TD metric (such as SQALE) and methods that provide proxies for TD metric with structural metrics (for example, similar to [53]). In their work, the authors are comparing the correlation between the SQALE metric and 20 structural object-oriented metrics from two metric suites [55, 56]. The case study is conducted on 20 open source projects. The results showed that several of the metrics can be used to quantify the TD since they strongly correlate with the SQALE metric. Although, the authors include high number of metrics in the comparison, they focus only on SQALE as the direct TD indicator and they investigate only the correlation between the metrics.

Zazworka et al. [57] studied and compared the human elicitation of TD with the automated TD identification. For the automated TD identification, the authors used code smell detection tools as well as common structural metrics.

Although different types of TD were considered, a detailed quantification of the results is not provided.

6. Conclusions

The goal of this paper was to compare three main techniques about TD identification that were proposed over time: (i) the Maintainability Index (MI), (ii) SIG TD models, and (iii) SQALE. We compared experimentally the three methods on a set of 20 Python projects.

Generally, all methods report an increasing trend of TD for the projects, but there are different patterns in the final evolution of the measurements time series. MI and SIG TD report generally more growing trends of TD compared to SQALE, which shows more periods of steady TD. MI is the methods that reports largely more repayments of TD compared to the other methods. SIG TD and MI are the models that show more similarity in the way TD evolves, while SQALE and MI are the less similar. The Granger causality for all projects and combination of methods shows that there is a limited dependency between the time series denoting the evolution of TD measurements. Still, we could find some relationships between SQALE and MI and SQALE and SIG TD models, in the sense that previous lags of SQALE time series could be used to improve prediction of other models in 1/3 of the projects.

Though exploratory in nature, by running the analysis on a different set of methods, we had findings comparable to previous research [11]. Seems there is limited overlap between the different methods for TD identification, probably due to the multifaced TD definition. One consequence being that practitioners adopting one or the other method might get different perceptions about the state of the projects. This can be seen from our analysis, when some models were reporting no changes in TD, while others larger repayments. One lesson learnt from the analysis is that using multiple methods can give more insights about the real presence of TD in a software project. By looking at the patterns from alternative methods, it is possible to understand better if TD is growing in an uncontrolled way.

Future works will go into extending the analysis to a larger scale and to get more explanatory insights, such as looking at relationships of models with common software quality metrics.

Data Availability

The mined datasets, experimental results with diagrams, and source code to support the findings of this study have been deposited in the Figshare repository (DOI: 10.6084/m9.figshare.11576040.v2) [44].

Disclosure

This paper is an extended version of Strečanský et al. [58]. The background about technical debt was extended, longer descriptions of the methods and results were provided with more analyzed projects, a new related works section, and extended references list.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

The work was supported by ERDF/ESF "CyberSecurity, CyberCrime and Critical Information Infrastructures Center of Excellence" (no. CZ.02.1.01/0.0/0.0/16_019/0000822).

References

[1] W. Cunningham, "The WyCash portfolio management system," *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2, pp. 29-30, 1993.
[2] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980.
[3] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *Journal of Systems and Software*, vol. 101, pp. 193–220, 2015.
[4] P. Kruchten, R. L. Nord, I. Ozkaya, and D. Falessi, "Technical debt: towards a crisper definition report on the 4th international workshop on managing technical debt," *ACM SIGSOFT Software Engineering Notes*, vol. 38, no. 5, pp. 51–54, 2013.
[5] B. Rossi, B. Russo, and G. Succi, "Modelling failures occurrences of open source software with reliability growth," in *Proceedings of the IFIP International Conference on Open Source Systems*, pp. 268–280, Springer, Notre Dame, IN, USA, May 2010.
[6] N. K. S. Roy and B. Rossi, "Towards an improvement of bug severity classification," in *Proceedings of the 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, pp. 269–276, IEEE, Verona, Italy, August 2014.
[7] B. Curtis, J. Sappidi, and A. Szynkarski, "Estimating the size, cost, and types of technical debt," in *Proceedings of the Third International Workshop on Managing Technical Debt (MTD'12)*, vol. 49–53, IEEE Press, Zurich, Switzerland, June 2012.
[8] K. Power, "Understanding the impact of technical debt on the capacity and velocity of teams and organizations: viewing team and organization capacity as a portfolio of real options," in *Proceedings of the 4th International Workshop on Managing Technical Debt (MTD)*, pp. 28–31, San Francisco, CA, USA, May 2013.
[9] E. Tom, A Aurum, and R. Vidgen, "An exploration of technical debt," *Journal of Systems and Software*, vol. 86, no. 6, pp. 1498–1516, 2013.
[10] D. Laribee, "Code cleanup—using agile techniques to pay back technical debt," 2009, https://msdn.microsoft.com/en-us/magazine/ee819135.aspx.
[11] N. Zazworka, A. Vetro', C. Izurieta et al., "Comparing four approaches for technical debt identification," *Software Quality Journal*, vol. 22, no. 3, pp. 403–426, 2014.
[12] S. McConnell, "Managing technical debt," Construx Software, Bellevue, WA, USA, 2008, https://www.construx.com/developer-resources/whitepaper-managing-technical-debt/ Technical Report.
[13] R. C. Martin, "A mess is not a technical debt," 2009, https://sites.google.com/site/unclebobconsultingllc/a-mess-is-not-a-technical-debt.
[14] D. Norton, "Messy code is not technical debt," 2009, http://docondev.com/blog/2009/08/messy-code-is-not-technical-debt.
[15] W. Cunningham, "Debt metaphor," 2009, https://www.youtube.com/watch?v=pqeJFYwnkjE.
[16] M. Fowler, "Technical debt," 2003, https://martinfowler.com/bliki/TechnicalDebt.html.
[17] J. Atwood, "Paying down your technical debt," 2009, https://blog.codinghorror.com/paying-down-your-technical-debt/.
[18] Y. Guo, R. Oliveira Spínola, and C. Seaman, "Exploring the costs of technical debt management—a case study," *Empirical Software Engineering*, vol. 21, no. 1, pp. 159–182, 2016.
[19] T. Theodoropoulos, M. Hofberg, and D. Kern, "Technical debt from the stakeholder perspective," in *Proceedings of the 2nd Workshop on Managing Technical Debt (MTD'11)*, pp. 43–46, ACM, New York, NY, USA, 2011.
[20] I. Gat, "Technical debt: technical debt: assessment and reduction," 2011, https://agilealliance.org/wp-content/uploads/2016/01/Technical_Debt_Workshop_Gat.pdf.
[21] C. Sterling, *Managing Software Debt: Building for Inevitable Change*, Addison-Wesley Professional, Boston, MA, USA, 1st edition, 2010.
[22] D. Rooney, "Technical debt: challenging the metaphor," *Cutter IT Journal*, vol. 23, no. 10, pp. 16–18, 2010.
[23] K. Schmid, "On the limits of the technical debt metaphor some guidance on going beyond," in *Proceedings of the 4th*

International Workshop on Managing Technical Debt (MTD), pp. 63–66, San Francisco, CA, USA, May 2013.

[24] M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, Boston, MA, USA, 1999.

[25] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas, "In search of a metric for managing architectural technical debt," in *Proceedings of the 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, pp. 91–100, Helsinki, Finland, August 2012.

[26] C. Camden, "Avoid getting buried in technical debt," 2011, https://www.techrepublic.com/blog/software-engineer/avoid-getting-buried-in-technical-debt/.

[27] J. Elm, "Design debt economics: a vocabulary for describing the causes, costs, and cures for software maintainability problems," 2009, http://www.startuplessonslearned.com/2009/07/embrace-technical-debt.html.

[28] R. Hilton, "When to work on technical debt," 2011, http://www.nomachetejuggling.com/2011/07/22/when-to-work-on-technical-debt/.

[29] B. Curtis, J. Sappidi, and A. Szynkarski, "Estimating the principal of an application's technical debt," *IEEE Software*, vol. 29, no. 6, pp. 34–42, 2012.

[30] E. da Silva Maldonado, E. Shihab, and N. Tsantalis, "Using natural language processing to automatically detect self-admitted technical debt," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1044–1062, 2017.

[31] J.-L. Letouzey, "The SQALE method for evaluating technical debt," in *Proceedings of the third International Workshop on managing technical debt (MTD)*, vol. 31–36, IEEE, Zurich, Switzerland, June 2012.

[32] R. Marinescu, "Assessing technical debt by identifying design flaws in software systems," *IBM Journal of Research and Development*, vol. 56, no. 5, pp. 9 : 1–9 : 13, 2012.

[33] A. Nugroho, J. Visser, and T. Kuipers, "An empirical model of technical debt and interest," in *Proceedings of the 2nd Workshop on Managing Technical Debt (MTD'11)*, vol. 1–8, ACM, New York, NY, USA, 2011.

[34] P. Oman and J. Hagemeister, "Metrics for assessing a software system's maintainability," in *Proceedings of the Conference on Software Maintenance*, pp. 337–344, Orlando, FL, USA, November 1992.

[35] O. Paul and J. Hagemeister, "Construction and testing of polynomials predicting software maintainability," *Journal of Systems and Software*, vol. 24, no. 3, pp. 251–266, 1994.

[36] S. Counsell, X. Liu, S. Eldh et al., "Re-visiting the "Maintainability Index" metric from an object-oriented perspective," in *Proceedings of the 41st Euromicro Conference on Software Engineering and Advanced Applications*, pp. 84–87, IEEE, Funchal, Portugal, August 2015.

[37] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability," in *Proceedings of the 6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*, pp. 30–39, Lisbon, Portugal, September 2007.

[38] C. Jones, "Backfiring: converting lines of code to function points," *Computer*, vol. 28, no. 11, pp. 87-88, 1995.

[39] LLC. Software Productivity Research, *SPR Programming Languages Table Ver. PLT2007c*, LLC. Software Productivity Research, Milford, CT, USA, 2007.

[40] A. Mayr, R. Plösch, and C. Körner, "A benchmarking-based model for technical debt calculation," in *Proceedings of the 14th International Conference on Quality Software*, pp. 305–314, IEEE, Dallas, TX, USA, October 2014.

[41] D. S. N Coghlan, "PEP 440–version identification and dependency specification," 2013, https://www.python.org/dev/peps/pep-0440/.

[42] S. Peter, "Dealing with software development technical debt," Master Thesis, Masaryk University, Brno, Czech Republic, 2019, https://is.muni.cz/auth/th/x0boz/master_thesis_digital.pdf.

[43] C. W. J. Granger, "Investigating causal relations by econometric models and cross-spectral methods," *Econometrica*, vol. 37, no. 3, pp. 424–438, 1969.

[44] S. Peter, S. Chren, and B. Rossi, "Comparing maintainability index, SIG method, and SQALE for technical debt identification," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, Brno, Czech Republic, March 2020.

[45] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, 2009.

[46] C. Izurieta, I. Griffith, D. Reimanis, and R. Luhr, "On the uncertainty of technical debt measurements," in *Proceedings of the International Conference on Information Science and Applications (ICISA)*, vol. 1–4, IEEE, Suwon, South Korea, June 2013.

[47] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and W. Anders, *Experimentation in Software Engineering*, Springer Science & Business Media, Berlin, Germany, 2012.

[48] C. Fernández-Sánchez, J. Garbajosa, A. Yagüe, and J. Perez, "Identification and analysis of the elements required to manage technical debt by means of a systematic mapping study," *Journal of Systems and Software*, vol. 124, pp. 22–38, 2017.

[49] W. N. Behutiye, P. Rodríguez, M. Oivo, and A. Tosun, "Analyzing the concept of technical debt in the context of agile software development: a systematic literature review," *Information and Software Technology*, vol. 82, pp. 139–158, 2017.

[50] I. Griffith, D. Reimanis, C. Izurieta, Z. Codabux, A. Deo, and B. Williams, "The correspondence between software quality models and technical debt estimation approaches," in *Proceeding of the Sixth International Workshop on Managing Technical Debt*, pp. 19–26, Victoria, BC, Canada, September 2014.

[51] F. R. Oppedijk, "Comparison of the sig maintainability model and the maintainability index" Ph.D. Dissertation. Master's thesis, University of Amsterdam, Amsterdam, Netherlands, 2008.

[52] I. Griffith, H. Taffahi, C. Izurieta, and D. Claudio, "A simulation study of practical methods for technical debt management in agile software development," in *Proceedings of the Winter Simulation Conference 2014*, pp. 1014–1025, Savanah, GA, USA, December 2014.

[53] Z. Li, P. Liang, P. Avgeriou, N. Guelfi, and A. Ampatzoglou, "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)*, vol. 119–128, Association for Computing Machinery, New York, NY, USA, 2014.

[54] M. V. Kosti, A. Ampatzoglou, A. Chatzigeorgiou, G. Pallas, I. Stamelos, and L. Angelis, "Technical debt principal assessment through structural metrics," in *Proceedings of the 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 329–333, Vienna, Austria, August 2017.

[55] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 4–17, 2002.

[56] W. Li and S. Henry, “Object-oriented metrics that predict maintainability,” *Journal of Systems and Software*, vol. 23, no. 2, pp. 111–122, 1993.

[57] N. Zazworka, R. O. Spínola, A. Vetro’, F. Shull, and C. Seaman, “A case study on effectively identifying technical debt,” in *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering (EASE’13)*, pp. 42–47, Association for Computing Machinery, New York, NY, USA, April 2013.

[58] E. Truyen, D. Van Landuyt, B. Lagaisse, and W. Joosen, “Performance overhead of container orchestration frameworks for management of multi-tenant database deployments,” in *Proceedings of the 35th ACM/SIGAPP Symposium on Applied Computing (SAC’20)*, vol. 5, Association for Computing Machinery, New York, NY, USA, April 2020.