applied sciences

MDPI

Article

Exploring Maintainability Index Variants for Software Maintainability Measurement in Object-Oriented Systems

Tjaša Heričko * and Boštjan Šumak

Faculty of Electrical Engineering and Computer Science, University of Maribor, Koroška Cesta 46, 2000 Maribor, Slovenia

* Correspondence: tjasa.hericko@um.si; Tel.: +386-2-220-7298

Abstract: During maintenance, software systems undergo continuous correction and enhancement activities due to emerging faults, changing environments, and evolving requirements, making this phase expensive and time-consuming, often exceeding the initial development costs. To understand and manage software under development and maintenance better, several maintainability measures have been proposed. The Maintainability Index is commonly used as a quantitative measure of the relative ease of software maintenance. There are several Index variants that differ in the factors affecting maintainability (e.g., code complexity, software size, documentation) and their given importance. To explore the variants and understand how they compare when evaluating software maintainability, an experiment was conducted with 45 Java-based object-oriented software systems. The results showed that the choice of the variant could influence the perception of maintainability. Although different variants presented different values when subjected to the same software, their values were strongly positively correlated and generally indicated similarly how maintainability evolved between releases and over the long term. Though, when focusing on fine-grained results posed by the Index, the variant selection had a larger impact. Based on their characteristics, behavior, and interrelationships, the variants were divided into two distinct clusters, i.e., variants that do not consider code comments in their calculation and those that do.

Keywords: software maintenance; maintainability measurement; Maintainability Index; software metrics; object-oriented software; Java

check for updates

Citation: Heričko, T.; Šumak, B. Exploring Maintainability Index Variants for Software Maintainability Measurement in Object-Oriented Systems. *Appl. Sci.* **2023**, *13*, 2972. https://doi.org/10.3390/app13052972

Academic Editor: Vito Conforti

Received: 9 February 2023
Revised: 20 February 2023
Accepted: 23 February 2023
Published: 25 February 2023

Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Software maintenance is a phase in the software life cycle that refers to modifications of a software system after delivery, aiming to provide support to the system [1]. During this phase, software evolves continuously through post-delivery enhancement and correction activities, including the identification and correction of faults, the improvement of software related to new functional and non-functional requirements, and the adaptation of software to changes in the environment [1,2]. According to existing studies [2–4], software maintenance is characterized as a highly cost-consuming phase in the software life cycle, as it has been estimated that up to 70% of time and resources related to a software project are allotted for maintenance activities. Hence, software maintenance is considered an important topic in software engineering research and practice. Aiming to understand and manage software under development and maintenance better, many research efforts have been focused on measuring maintainability, a software quality factor that refers to the degree of effectiveness and efficiency of software to be modified, to correct faults, improve with respect to alterations in the requirements, and adapt to changes in the environment [1,5]. Thus, measuring the maintainability of a software system can estimate the ease with which the system can be maintained.

Maintainability is an external quality attribute. Hence, it requires being measured and quantified indirectly while relying on internal attributes that can be measured from software

directly [6]. In addition, several different definitions and understandings of maintainability in the context of a software system's quality exist in the literature [6,7], resulting in a variety of measures and models defined to aid software practice and research with measuring and quantifying software maintainability. In the early 1990s, the Maintainability Index was proposed, a quantitative single-valued indicator of software maintainability [8]. A higher value of the Index indicates higher maintainability, which suggests that the software is easier to maintain, whereas a lower value of the Index indicates lower maintainability, which suggests that the software is more difficult to maintain. The Index is calculated by taking into account software characteristics that are believed to impact the maintainability. For example, the original three-metric Maintainability Index is computed by considering three traditional size and complexity software metrics extracted from source code—the Halstead's Effort metric, the Cyclomatic Complexity metric, and the Lines of Code metric—metrics that reflect the mental effort required to maintain software, the complexity of the code, and the size of the software [8,9]. In the last three decades, several variants of the Index were defined in an effort for the Index to better capture the maintainability characteristics of software, to ease calculation, or to provide results in a clearer manner [8–17]. The variants differ mainly based on the different emphases on the software characteristics derived from the code, which is reflected by the selection of the software metrics included in the equation and/or coefficients of the polynomial equation for computing the Index. Yet, all of them have the same objective—to indicate maintainability, i.e., the ease of software to be maintained on the basis of a software system's code.

In the last few decades, the object-oriented paradigm has emerged as one of the most-widely used paradigms for modeling and developing software. Consequently, the maintainability of object-oriented software has become a significant topic of interest for both software researchers and practitioners. Although concerns regarding the appropriateness of the Index for object-oriented languages have been expressed because the measure was initially proposed for systems developed in procedural programming languages [15,18–21], the Index is frequently used in maintainability research for object-oriented software. For instance, the Index is used for monitoring the maintainability of a software system over time to ensure high-quality software products, to guide and support software-related decision-making processes or characterize the software's maintainability evolution [10,22–27], to explore the relationship between maintainability and design or code metrics [14–16,20,28,29], to detect technical debt [30], and recently, the Index has been used as a measure of maintainability in various machine learning prediction models [31–35].

As discussed above, several variants of the Index exist and are often used interchangeably. This is also the case with the aforementioned studies, as the choice of the variant used for the maintainability measurement of object-oriented software varies across them. Naturally, it is to be expected that maintainability assessment based on different variants computed using different equations may yield different results when subjected to the same software. The main advantage of the Index as a single-value measure of maintainability is its ability to be able to compare software in the context of others [9,36], either to its previous versions or to completely different systems. Hence, besides possible differences in the value of different Index variants when applied to measure the maintainability of a single software, it is important to determine how the use of different Index variants influences the relationship between the maintainability of several different software. If the results of Index variants would prove to be significantly different, this could mean that decisions and conclusions based on the chosen variant might be different if another variant was utilized for maintainability assessment instead. Contrariwise, if the results of variants are shown to be similar, this could indicate that the choice of the variant does not significantly affect the end results. This has not been researched previously; thus, these assumptions cannot be verified without proper empirical investigations.

The objectives of this work were twofold: first, to highlight distinct variants of the Maintainability Index employed to measure maintainability in object-oriented systems and assess the results of variants when applied to measure the maintainability of the same

subject software; second, to research the results of the variants when applied to measure the maintainability of a software system to compare the system with other systems.

To meet the above-specified objectives, the following two research questions (RQs) were formulated:

RQ1 How do different Maintainability Index variants perform when utilized for the maintainability measurement of a single object-oriented software system?

RQ2 How do different Maintainability Index variants perform when utilized for the maintainability measurement of an object-oriented software system in the context of other systems?

- RQ2.1 How do different Maintainability Index variants perform when utilized for maintainability measurement between different software systems?
- RQ2.2 How do different Maintainability Index variants perform when utilized for maintainability measurement between versions of the same software system?

An experiment was conducted on data collected from the source code of 45 open-source software systems based on the Java programming language. For 42 software systems, only the last released version was included in the experiment, while for the remaining 3 systems, all versions from the release history were included. The Maintainability Index variants were computed from software metrics extracted from the code of each system by the JHawk tool. To answer RQ1, an evaluation and analysis of the gathered data were performed for one version of each included software system, focusing on the differences between the Index variants. To answer RQ2, the results were analyzed from two perspectives: between versions of the same software system, where three software systems with their versions were considered, and between different software systems, where the last version of each included software system was considered.

In summary, the key contributions of this paper are: (1) a review of Maintainability Index variants utilized for maintainability measurement in object-oriented software; (2) an empirical investigation on how Maintainability Index variants perform when utilized for the maintainability measurement of a single object-oriented system; (3) an empirical investigation on how Maintainability Index variants perform when utilized for the maintainability measurement of an object-oriented software system in the context of other systems, i.e., other versions of the same software system and completely different software systems.

The rest of the paper is structured as follows. Section 2 provides the background on object-oriented software maintainability measurements needed to follow this paper. Related work is reviewed and discussed in Section 3. Section 4 presents and explains the experiment's procedure. The experiment results are reported in Section 5. The paper continues by analyzing and discussing the obtained results in Section 6. Finally, Section 7 provides conclusions and outlines possible related open research topics and areas.

2. Background

2.1. Object-Oriented Software

The object-oriented paradigm is one of the dominant programming paradigms for modeling and developing software. Programming languages, designed mainly for object-oriented programming, namely Java, C#, and Python, are currently widely used and adapted by organizations and software developers [37,38]. Due to the emerging use of object orientation in the last few decades, concerns regarding the effective measurement and management of maintainability in object-oriented software have been raised. Object orientation as a style of constructing software differs in several ways from other programming paradigms, for instance the procedure-oriented paradigm. It is based on the notion of objects as self-contained units that have an identity, state, and behavior [39,40]. Object-oriented design concepts, in particular encapsulation, inheritance, and polymorphism, can affect software quality characteristics, including maintainability [40,41]. Hence, in the paper, we only focused on maintainability measures originally proposed for object-oriented software. Additionally, we included measures that were later adopted, tested, and accepted

to be capable of analyzing software using the object-oriented programming paradigm as well.

2.2. Maintainability Measurement in Object-Oriented Software Systems

The maintainability of a system is a software quality factor that reflects the ease with which a system can undergo various modifications in the software maintenance phase [5], namely corrective, preventive, adaptive, additive, and perfective [1]. Providing insight into the maintainability of software helps to gain a better understanding of the software and its ability to be corrected or enhanced. This consequently helps to determine maintenance costs, time, and effort, to manage and plan software and software-related resources efficiently, and to identify areas for improvement [9,36]. Since the maintainability of software is an external quality attribute that cannot be measured directly, various maintainability measures and models are being employed for measuring maintainability [6].

To quantify the software maintainability of any system with a single value, Oman and Hagemeister [8] proposed the Maintainability Index. The Index is given as a polynomial expression based on software metrics that describe the characteristics of software. The polynomial was constructed through a series of experiments using regression analysis. Subjective assessment of the maintainability evaluated by software practitioners maintaining the code of selected subject software systems was used as the dependent variable and software metrics extracted from the code of the subject software as independent variables. The researchers identified the smallest set of easily obtained source code software metrics with which the software maintainability can be estimated reasonably accurately. Hence, with the constructed regression equation, the Index can be calculated for any software based on software size and complexity metrics extracted from source code [8,9]. The resulting single value indicates the maintainability of that system. A software system with a value of the Index below 65 is considered to have low maintainability, a value between 65 and 85 medium maintainability, and a value above 85 high maintainability. Thus, in general, a high value of the Index represents higher software maintainability, whereas a low value of the Index represents lower maintainability [42]. The Maintainability Index was originally proposed and validated based on observations and adaptations on a small set of software systems developed in procedural programming languages. As a result, researchers [15,18–21] have highlighted a number of potential limitations of the Index when applied to the object-oriented paradigm: (1) the coefficients in the formula for the computation of the Index were obtained using regression analysis on software developed in C and Pascal, which might have different maintainability characteristics than current object-oriented programming languages, such as Java; (2) the Index only considers traditional software metrics and omits object-oriented metrics related to cohesion, coupling, polymorphism, and inheritance, which are considered as important characteristics of object-oriented software; (3) threshold values for distinguishing between low, medium, and high maintainability were not validated for object-oriented software. Despite these concerns, the Index has reached general acceptance and is frequently used in maintainability research for object-oriented software. Studies conducting literature reviews on software maintainability [6,7,43–45] report that the Index is one of the most-commonly used measure of maintainability. In addition, the Index is widely adopted in practice as the calculation of the Index is available within development environments, e.g., Visual Studio [12], as well as in popular metrics collection tools, e.g., JHawk [46], Radon [47], CMT++, and CMTJava [48].

Another measure of software maintainability is the change metric defined by Li and Henry [49]. This measure estimates maintainability by counting the number of code lines changed per class, where a line change can be an addition or a deletion. The authors further investigated the relationship between a set of metrics collected from code (two size metrics and several object-oriented metrics) and maintainability, showing that maintenance effort can be predicted from the selected metrics. A high value of the change metric represents low maintainability, while a low number represents high maintainability [6].

Malhotra and Khanna [50] and Elish et al. [51] defined software maintainability in the form of class-level change proneness. It represents the possibility that a change, i.e., addition, deletion, or modification, occurs in a source code of a class after the release of a software system. Maintainability change proneness is quantified as a binary metric: its value is "Yes" if the change occurs or "No" if the change does not occur [6].

Furthermore, researchers have proposed several maintainability measurement and prediction models based on capturing maintenance performance with cost, time, or effort spent in the maintenance process [3,52–55]. Granja-Alvarez et al. [3] proposed a model for estimating maintenance costs based on the time required to perform maintenance actions. The estimated maintenance cost is computed as the sum of the costs of three maintenance actions, namely understanding, modifying, and testing. Bandi et al. [52] validated an approach to predicting the time consumed to execute maintenance tasks using the design complexity of a system, measured by the following three object-oriented design complexity metrics: Interaction Level, Interface Size, and Operation Argument Complexity. Larger quantities of maintenance costs or time spent indicate lower maintainability, while smaller quantities of costs or time spent indicate higher maintainability [6]. Fioravanti and Nesi [53] presented a model for adaptive maintenance effort estimation of object-oriented systems, where effort spent is computed as the sum of effort spent due to understanding, addition, and deletion of the source code. To estimate the maintenance effort of each operation, several metrics were considered, among which class complexity and size metrics proved to be the most-suitable. Another model for estimating adaptive maintenance effort, based on the number of lines of code changed or the number of operators changed, was proposed by Hayes et al. [54]. De Lucia et al. [55] proposed an effort estimation model for corrective maintenance, where effort is calculated using the size of the system to be maintained and the number of three distinct types of maintenance tasks. The resulting effort metric measures the actual effort spent in each phase of the maintenance process (e.g., define, analyze, design, produce, or implement phase) or in the whole maintenance period for corrective activities. For both corrective and adaptive maintenance effort, a low measurement of effort represents high maintainability, whereas a high measurement indicates low maintainability [6].

Another way of assessing maintainability is based on the subjective evaluation of software systems by experts [56–59]. In this case, maintainability is expressed on an ordinal scale using expert opinion, for instance as poor, average, good, and very good [56], or as very low, low, medium, high, and very high [57,58], or as very easy, easy, medium, high, and very high [59].

2.3. Maintainability Index Variants

With extensive use of the Maintainability Index in software research and practice, the Index has been fine-tuned over time to better represent the maintainability characteristics of software systems, to ease the calculation, or to aid the interpretation of the results. This resulted in several defined variants of the Index. The key differences between the variants lie in the selection of the software metrics used and/or the amount of influence the selected software metrics have on maintainability.

The original three-metric Maintainability Index (further referred to as $MI_{orig.}$) proposed by Oman and Hagemeister [8] is defined as:

$$MI_{orig.} = 171 - 3.42 \times \ln(aveE) - 0.23 \times aveV(g) - 16.2 \times \ln(aveLOC) \quad (1)$$

where *aveE* is the average Halstead's Effort per module, *aveV(g)* is the average Cyclomatic Complexity per module, and *aveLOC* is the average Lines of Code per module. Besides the three-metric Index that captures the size and complexity aspects of the code, Oman and Hagemeister [8] proposed a four-metric Maintainability Index that additionally includes a self-explanatory aspect of the code. This was included in the equation by capturing the quantity of comments in the source code in a way that the presence of comments is considered to improve maintainability. The comment part of the equation first included the average Number of Lines of Comments per module. This metric was later criticized

as being too heavily skewed by large comment blocks, which has the potential to increase the value of the resulting Index excessively. Thus, to mitigate this thread of the Index being overly sensitive to comments, a more subtle Index variant was proposed using a ratio between comment lines and code lines instead of an absolute number of comment lines in the code [8–10]. The original four-metric Maintainability Index ($MI_{orig.(CM)}$) with the comment part, which limits the impact comments can have on maintainability, is calculated as

$$MI_{orig.(CM)} = 171 - 3.42 \times \ln(aveE) - 0.23 \times aveV(g) - 16.2 \times \ln(aveLOC) + 50 \times \sin(\sqrt{2.4 \times perCM}) \quad (2)$$

where *aveE* is the average Halstead's Effort per module, *aveV(g)* is the average Cyclomatic Complexity per module, *aveLOC* is the average Lines of Code per module, and the additional *perCM* is the average Percent of Lines of Comments per module. Due to criticisms regarding the non-monotonic nature of the Halstead's Effort metric, the Index was later reconstructed using Halstead's Volume metric instead since the difference between the two was not found to be statistically significant [9,10]. This resulted in the improved three-metric Maintainability Index ($MI_{impr.}$) calculated as

$$MI_{impr.} = 171 - 5.2 \times \ln(aveV) - 0.23 \times aveV(g) - 16.2 \times \ln(aveLOC) \quad (3)$$

where *aveV* is the average Halstead's Volume per module, *aveV(g)* is the average Cyclomatic Complexity per module, and *aveLOC* is the average Lines of Code per module. Again, the reconstructed Index was also proposed with the comment part [9,10], resulting in the improved four-metric Maintainability Index ($MI_{impr.(CM)}$) defined as

$$MI_{impr.(CM)} = 171 - 5.2 \times \ln(aveV) - 0.23 \times aveV(g) - 16.2 \times \ln(aveLOC) + 50 \times \sin(\sqrt{2.4 \times perCM}) \quad (4)$$

where *aveV* is the average Halstead's Volume per module, *aveV(g)* is the average Cyclomatic Complexity per module, *aveLOC* is the average Lines of Code per module, and the additional *perCM* is the average Percent of Lines of Comments per module. A study by Revilla [60] revealed that there is a strong correlation between Lines of Code and Halstead's Volume, as well as between Lines of Code and Cyclomatic Complexity. Based on these findings, Najm [11] substituted Halstead's Volume part and the Cyclomatic Complexity part in the Index equation with the respective calculations of these two parts using Lines of Code, which allows quicker and easier computation of the Index. The author built upon the improved Index and proposed the Maintainability Index computed using only Lines of Code (MI_{LOC}) as follows:

$$MI_{LOC} = 171 - 5.2 \times \ln(45 \times aveLOC - 428) - 0.23 \times (0.22 \times aveLOC - 1.9) - 16.2 \times \ln(aveLOC) \quad (5)$$

where *aveLOC* is the average Lines of Code per module. The JHawk product introduced two additional Index variations that use the Number of Statement metric instead of the Lines of Code metric. The former is believed to be a better measure of software size than the latter [13]. Thus, the JHawk three-metric Maintainability Index (MI_{JH}) is calculated as

$$MI_{JH} = 171 - 5.2 \times \ln(aveV) - 0.23 \times aveV(g) - 16.2 \times \ln(aveNOS) \quad (6)$$

where *aveV* is the average Halstead's Volume per module, *aveV(g)* is the average Cyclomatic Complexity per module, and the substituted *aveNOS* is the average Number of Statements

per module. Again, the Index was also defined with the comment part, resulting in the JHawk four-metric Maintainability Index ($MI_{JH(CM)}$) as follows:

$$MI_{JH(CM)} = 171 - 5.2 \times \ln(aveV) - 0.23 \times aveV(g) - 16.2 \times \ln(aveNOS) + 50 \times \sin(\sqrt{2.4 \times perCM}) \quad (7)$$

where *aveV* is the average Halstead's Volume per module, *aveV(g)* is the average Cyclomatic Complexity per module, *aveNOS* is the average Number of Statements per module, and *perCM* is the average Percent of Lines of Comments per module. The results of the presented Indices range from an unbounded negative number to 171. For the result of the Index to be bound between 0 and 100, which provides a clearer and more useful interpretation, Visual Studio derived an Index variant based on the improved Index with the sole difference of using a shifted scale [61]. The Maintainability Index defined by Visual Studio (MI_{VS}) is calculated as

$$MI_{VS} = max\left(0, 100\frac{171 - 5.2 \times \ln(aveV) - 0.23 \times aveV(g) - 16.2 \times \ln(aveLOC)}{171}\right) \quad (8)$$

where *aveV* is the average Halstead's Volume per module, *aveV(g)* is the average Cyclomatic Complexity per module, and *aveLOC* is the average Lines of Code per module.

Each of the presented variants represents a slightly different approach to measuring maintainability by capturing different software-related characteristics. Table 1 provides a summary of the software metrics required to compute each variant and the corresponding software aspects that these metrics measure. As for measuring code complexity, both Halstead's Effort and Halstead's Volume measure computational complexity, while McCabe's Cyclomatic Complexity measures logical complexity. Lines of Code and Number of Statements measure software size. Percent of Lines of Comments measures the self-explanatory nature of the code base through documentation, i.e., code comments.

Table 1. The inclusion of software metrics for the computation of Index variants and software aspects the metrics measure.

	Computational Complexity		**Logical Complexity**	**Software Size**		**Code Documentation**
Maintainability Index Variant	**Halstead's Effort**	**Halstead's Volume**	**McCabe's Cyclomatic Complexity**	**Lines of Code**	**Number of Statements**	**Percent of Lines of Comments**
$MI_{orig.}$	X		X	X		
$MI_{orig.(CM)}$	X		X	X		X
$MI_{impr.}$		X	X	X		
$MI_{impr.(CM)}$		X	X	X		X
MI_{LOC}				X		
MI_{JH}		X	X		X	
$MI_{JH(CM)}$		X	X		X	X
MI_{VS}		X	X	X		

Mark "X" is used to indicate the inclusion of a software metric for the computation of a specific Index variant.

Besides the above-presented Index variants, several others were proposed in the literature incorporating other software metrics that measure additional software aspects, e.g., cohesion and coupling; yet, they are not widely used. Misra [16] proposed the Index based on design/code-level metrics, including the average Method Size, Program Length, Control Density, Depth of Inheritance Tree, and Method Hiding Factor. Kaur and Singh [14] proposed the Index using package metrics, namely size, complexity, cohesion, and nesting metrics. Kaur et al. [15] proposed two Indices based on object-oriented metrics, such as Lack of Cohesion in Methods, Message Passing Coupling, Response for a Class, Weighted Method per Class, Number of Commands, Number of Constructors, and Number of Cyclic

Dependencies. Madhwaraj [17] proposed the Index using the object-oriented Martin's metrics suite, where Distance From the Main Sequence, Afferent Coupling, Number of Concrete Classes, and Efferent Coupling are needed for computation. The author also proposed the Index using the object-oriented Chidamber and Kemerer metrics suite, where Coupling Between Objects and Response For a Class are considered for computation.

The evolution of the initially proposed Maintainability Index into numerous variants has resulted in a varying usage of Index variants in the context of maintainability assessment. This issue has already been noted by systematic literature reviews conducted in the research field [43,45]. The most widely used variants and their example usage in the literature regarding object-oriented software are summarized in Table 2. The table highlights that different Index variants are used across the research community, while using software based on various object-oriented programming languages. The Index variants are used for several purposes, including maintainability monitoring [10,22–27,62], exploration of the relationship between maintainability and software metrics [14–16,20,28,29,63], characterization of architectural design patterns in view of maintainability [64–66], investigation of the impact of community patterns on software maintainability [67], technical debt detection [30], and machine-learning-based fault, change proneness, and maintainability trend prediction [31–35].

Table 2. Overview of studies using different Index variants.

Maintainability Index Variant	Example Study Using the Variant	Programming Language of Object-Oriented Software Systems Used in the Study
$MI_{orig.}$	[10] [34,64]	C++ Java
$MI_{orig.(CM)}$	[10] [14,20]	C++ Java
$MI_{impr.}$	[27] [22] [15,27,29,33,63,67,68] [67]	C# C++ Java Python
$MI_{impr.(CM)}$	[16] [28] [65]	C++ Java Python
MI_{LOC}	[26]	C#
MI_{JH}	[31,32,62]	Java
$MI_{JH(CM)}$	[31,32]	Java
MI_{VS}	[23,26,66] [24,25,35] [30]	C# Java Python

3. Related Work

In the literature, the Maintainability Index has been used as a measure of maintainability extensively researched from various perspectives. Although numerous measures have been proposed in the literature for assessing software quality factors, few studies have focused on comparing the Index with other maintainability or software quality measures. A complete overview of related studies is presented in Table 3.

Table 3. Overview of related studies.

Study	Compared Software Measures	Comparison Objective	Subject Software Systems	Research Findings
Kaur and Singh [14]	Maintainability Index ($MI_{orig.(CM)}$), proposed maintainability metric	Validating a new measure	Releases of three Java software systems	N/A
Kaur et al. [15]	Maintainability Index ($MI_{impr.}$), change metric	Evaluating the maintainability prediction power of the Index for object-oriented software on the class level	Two releases of a Java software system	There is a slight inverse relationship between the measures
Kencana et al. [26]	Maintainability Index (MI_{VS}, MI_{LOC})	Comparing the Index calculations from two frameworks on a software and component level	A C# software system	The difference between calculations is negligible
Madhwaraj [17]	Maintainability Index ($MI_{impr.(CM)}$), proposed maintainability metric	Validating a new measure	Releases of four Java software systems	N/A
Misra [16]	Maintainability Index ($MI_{impr.(CM)}$), proposed maintainability metric	Validating a new measure	Fifty C++ software systems	N/A
Najm [11]	Maintainability Index ($MI_{impr.}$), proposed maintainability metric	Validating a new measure	Six C++ software systems	N/A
Papamichail and Symeonidis [25]	Maintainability Index ($MI_{VS.}$), proposed maintainability metric	Validating a new measure	Three releases of a Java software system	N/A
Sjøberg et al. [19]	Maintainability Index ($MI_{impr.(CM)}$), two code smells (Feature Envy, God Class), a set of structural metrics, system size metrics, maintenance effort	Investigating the consistency of software maintenance metrics at the system level	Four Java software systems	The measures are not mutually consistent for evaluating maintainability
Strečanský et al. [30]	Maintainability Index (MI_{VS}), SIG method, SQALE analysis	Comparing methods for software technical debt identification on the between-release level	Releases of twenty Python software systems	The Index and SIG method show more similarity, while the Index and SQALE analysis show less

N/A = not applicable.

Some studies have compared the Index with alternative software quality measures with the aim of assessing existing measures. Sjøberg et al. [19] researched consistency at a software system level among a set of software maintenance measures, including the Maintainability Index, code smells, structural metrics related to coupling, cohesion, size, and inheritance, software size metrics, and maintenance effort measured by the time spent on a maintenance task and the number of changes completed in the course of the task on the subjected software systems. The empirical research on four functionally equivalent web-based systems, primarily implemented in the Java programming language, revealed that these measures are not mutually consistent for evaluating maintainability. The inverted value of the Index had the highest positive correlation with the inverted value of the Depth of Inheritance Tree metric ($r_s = 1$), coupling measure ($r_s = 0.8$), and Feature Envy code smell type ($r_s = 0.8$). It had the highest negative correlation with the inverted value of the cohesion measure ($r_s = -1$), Lines of Code metric ($r_s = -0.6$), and maintenance effort measured by the average hours spent on the maintenance task ($r_s = -0.6$). Kaur et al. [15]

compared the Maintainability Index with the change metric by calculating the differences between individual classes of two versions of an open-source software written in Java. The results showed a low negative correlation ($r = -0.32$), suggesting a slight inverse relationship between the measures. Kencana et al. [26] compared the Maintainability Index calculation used in Visual Studio from two different frameworks; from built-in Code Metric Analysis and the Microsoft CodeLens Code Health Indicator extension. An experiment on a C# software system demonstrated a negligible difference between calculations. Strečanský et al. [30] compared three techniques for software technical debt identification, namely the Maintainability Index, SIG method, and SQALE analysis, on releases of 20 open-source Python libraries. The results showed that each method gives a slightly different perception of how technical debt evolves. The Index and SIG method show more similarity regarding trends of technical debt changes between releases ($r_{Med.} = 0.67$), while the Index and SQALE analysis show less ($r_{Med.} = 0.57$).

Some studies have compared the well-established Maintainability Index variant with alternative software quality measures with the aim of validating new measures. Kaur and Singh [14] compared their proposed maintainability metric with the original four-metric Maintainability Index. Najm [11] compared the proposed metric with the improved three-metric Maintainability Index. Misra [16] and Madhwaraj [17] compared their proposed maintainability metric with the four-metric improved Maintainability Index. Papamichail and Symeonidis [25] compared the proposed maintainability evaluation with the Maintainability Index defined by Visual Studio.

Although there are some similarities with the existing work, the lack of a comprehensive approach to comparing multiple variations of the Index makes our work the first of its kind in the field. Firstly, our study differs in the inclusion of multiple commonly used Index variants in the comparison, while most studies, except for [26], did not include more than one variant. Instead, they focused on the comparison with other software measures. This limitation of the current works prevents the works from presenting a clear picture regarding the consistency of the Index variants for software maintainability measurement. Secondly, most studies conducted the comparison on a relatively small set of software systems, while we conducted experiments on a more extensive set of subject software systems. Furthermore, while the existing works mainly focused on the comparative analysis of either one subject or between multiple subjects, we provide analyses in both contexts.

4. Research Method

To achieve the research objectives, experiments were conducted on Java open-source software systems. A total of 45 object-oriented software systems were randomly selected for the study. Projects varying in maturity, popularity, domain, and size were included to ensure a diverse sample. To answer RQ1 and RQ2.1, the most recently released version at the time of conducting the experiments for each of the 45 software systems under study was used. Table 4 provides an overview of the software systems studied. The source code for each included subject was retrieved from the GitHub repository specified in the table. To address RQ2.2, three software systems, varying in popularity and source code characteristics, were selected as case studies, with all available versions of their release histories included in the experiment. Table 5 presents these selected software systems and their releases.

The Maintainability Index was calculated for each software system under study using seven variants: $MI_{orig.}$, $MI_{orig.(CM)}$, $MI_{impr.}$, $MI_{impr.(CM)}$, MI_{LOC}, MI_{JH}, and $MI_{JH(CM)}$. These variants were chosen because software researchers and practitioners most commonly utilize them to assess the maintainability of object-oriented software. The software metric values needed for the computations were derived from the source codes of the subject software using JHawk 6.1.4, a Java software metric collection tool. Then, the seven variants were calculated using the formulas listed in Section 2.3. When calculating, the metric values were averaged per class since this is a common practice when calculating the Index at the system level in an object-oriented software ecosystem. In Table 6, descriptive statistics of

the collected software metric values of 45 studied software systems, averaged per class, are reported. In addition, these data are presented visually with boxplots in Figure 1. The resulting Index variant values are available in Appendix A. In Table A1, Index variant values relevant for RQ1 and RQ2.1 are reported. In Tables A2–A4, Index variant values relevant for RQ2.2 are reported for *CS1*, *CS2*, and *CS3*, respectively.

Table 4. Overview of subject software systems included in the study (RQ1 and RQ2.1).

Software System	Code Repository [a]	GitHub Stars [b]	Release	Number of Classes	Lines of Code	Lines of Comments
S1. Activiti	*Activiti/Activiti*	8.9k	7.4.0	3,130	181,012	67,996
S2. Angry IP Scanner	*angryip/ipscan*	2.9k	3.8.2	382	12,971	2878
S3. Apache Ant	*apache/ant*	337	1.10.12	1817	139,425	104,736
S4. Apache Commons Codec	*apache/commons-codec*	366	1.15	190	23,873	15,319
S5. Apache Commons CSV	*apache/commons-csv*	291	1.9.0	56	8314	3726
S6. Apache Commons DBCP	*apache/commons-dbcp*	286	2.9.0	189	32,110	11,173
S7. Apache Commons Lang	*apache/commons-lang*	2.3k	3.12.0	916	78,468	58,556
S8. Apache HttpClient	*apache/httpcomponents-client*	1.2k	5.1.3	1179	72,840	29,967
S9. Apache PDFBox	*apache/pdfbox*	1.7k	2.0.26	1587	167,686	85,153
S10. Apache POI	*apache/poi*	1.5k	5.2.2	4534	399,492	179,826
S11. Arduino	*arduino/Arduino*	13.1k	1.8.19	429	25,911	10,262
S12. Art of Illusion	*ArtOfIllusion/ArtOfIllusion*	61	3.2.0	901	118,268,809	18,676
S13. AssertJ	*assertj/assertj-core*	2.2k	3.23.1	5359	201,769	158,914
S14. Caffeine	*ben-manes/caffeine*	12.1k	3.1.1	925	55,991	15,328
S15. cglib	*cglib/cglib*	4.5k	3.3.0	498	15,181	4353
S16. DITA Open Toolkit	*dita-ot/dita-ot*	322	3.7.2	499	49,117	12,357
S17. EasyMock	*easymock/easymock*	776	4.3	336	14,757	6930
S18. Ehcache	*ehcache/ehcache3*	1.8k	3.10.1	2374	127,468	41,302
S19. FastJSON	*alibaba/fastjson*	24.9k	1.2.83	6295	182,197	13,922
S20. GeOxygene	*IGNF/geoxygene*	31	1.9	2767	241,496	114,263
S21. h2database	*h2database/h2database*	3.4k	2.1.214	1657	238,082	69,448
S22. Hibernate ORM	*hibernate/hibernate-orm*	5.1k	5.6.11	15,382	798,556	191,614
S23. iText7	*itext/itext7*	1.3k	7.2.3	3063	292,122	159,854
S24. JabRef	*JabRef/jabref*	2.7k	5.7	1935	131,891	15,927
S25. Jajuk	*jajuk-team/jajuk*	41	11.0	1043	67,423	47,767
S26. JasperReports	*TIBCOSoftware/jasperreports*	702	6.20.0	3720	319,275	150,282
S27. javaGeom	*dlegland/javaGeom*	44	0.11.3	376	32,577	17,992
S28. Java Hamcrest	*hamcrest/JavaHamcrest*	2k	2.2	242	6980	3532
S29. Jenkins	*jenkinsci/jenkins*	19.4k	2.366.1	3703	172,272	77,378
S30. JFreeChart	*jfree/jfreechart*	888	1.5.3	1056	136,664	127,798
S31. JFreeSVG	*jfree/jfreesvg*	251	5.0.3	24	3930	3,164
S32. JGraphT	*jgrapht/jgrapht*	2.2k	1.5.1	1261	128,742	60,731
S33. JMeter	*apache/jmeter*	6.4k	5.5	1680	146,352	68,114
S34. Joda-Time	*JodaOrg/joda-time*	4.8k	2.11.1	557	88,232	44,692
S35. jsoup	*jhy/jsoup*	9.7k	1.15.3	244	26,125	5708
S36. JUnit4	*junit-team/junit4*	8.3k	4.13.2	1469	31,242	7460
S37. JUnit5	*junit-team/junit5*	5.4k	5.9.0	2314	81,910	35,277
S38. Mockito	*mockito/mockito*	13.3k	4.7.0	1948	58,710	19,848
S39. MPAndroidChart	*PhilJay/MPAndroidChart*	35.5k	3.1.0	300	24,370	8930
S40. PowerMock	*powermock/powermock*	3.9k	2.0.9	1220	37,852	20,675
S41. SLF4J	*qos-ch/slf4j*	2k	2.0.0	260	13,397	10,046
S42. Spring	*spring-projects/spring-framework*	49k	5.3.22	14,088	675,735	346,280
S43. Spring Boot	*spring-projects/spring-boot*	63k	2.7.3	10,431	351,935	164,354
S44. TestNG	*cbeust/testng*	1.8k	7.6.1	2444	91,953	9616
S45. YamlBeans	*EsotericSoftware/yamlbeans*	525	1.15	186	8485	894

[a] GitHub code repositories are available at https://github.com/ (accessed on 30 August 2022). [b] Data gathered on 30 August 2022.

Table 5. Overview of subject software systems included in the study (RQ2.2).

			Number of Classes			Lines of Code			Lines of Comments		
Case Study	**Software System**	**Releases**	**Avg.**	**Med.**	**Std.**	**Avg.**	**Med.**	**Std.**	**Avg.**	**Med.**	**Std.**
CS1	*S13.* AssertJ	63 (1.0.0–3.23.1)	3346.9	3065	1133	125,175.9	112,809	44,201.9	94,521.7	82,938	35,557.8
CS2	*S14.* Caffeine	61 (1.0–3.1.1)	760.6	7836	117.1	45,474.5	46,574	7364.8	12,756	13,204	1411.4
CS3	*S34.* Joda-Time	52 (0.9–2.11.1)	547.9	557	72.2	80,046.7	85,490	13,880.8	42,200.2	44,552	5451.7

Avg. = average, Med. = median, Std. = standard deviation.

Table 6. Descriptive statistics of software metric values averaged per class.

	Halstead's Effort per Class	**Halstead's Volume per Class**	**McCabe's Cyclomatic Complexity per Class**	**Lines of Code per Class**	**Number of Statements per Class**	**Percent of Lines of Comments**
Avg.	54,201	2458	12.1	76.0	54.6	43.4
Med.	37,166	2164	11.3	64.6	47.9	44.9
Std.	58,992	1491	6.81	41.0	29.9	20.4
Min.	2957	377	3.64	21.3	13.9	7.64
Max.	288,226	5693	33.3	170	123	93.5

Avg. = average, Med. = median, Std. = standard deviation, Min. = minimum, Max. = maximum.

Next, data analysis on the acquired data was performed using IBM SPSS Statistics 28 and Python 3.7.6. with the matplotlib, NumPy, pandas, SciPy, seaborn, and scikit-learn libraries. A variety of data analysis methods (i.e., descriptive statistics, hypothesis testing, correlation analysis, multidimensional scaling, cluster analysis, and trend analysis) were used to answer the research questions, which allowed us to evaluate and compare the consistency of several variants of the Index in measuring the maintainability of object-oriented software systems on a large and diverse set of Java software in different contexts.

To answer RQ1, descriptive statistics were initially used to describe the values of each Index variant and to gain a general understanding of the distribution of the data. This allowed us to evaluate how the different variants capture the notion of maintainability when applied to measure the maintainability of the same software system and to compare the variants in terms of their absolute values. Next, focusing on identifying any significant differences between the variants, the pairwise differences between Index variants were assessed. Inferential statistics were employed to determine if there was a statistically significant difference in the maintainability measurements of the two Index variants. To verify the assumptions of normality for the differences between the two groups under study, the Shapiro–Wilk Test was utilized because the sample size for the data of RQ1 was small ($n \leq 50$). When the test showed that the normality assumption was met, i.e., that the differences between the two groups followed a normal distribution, a Paired-Samples *t*-Test was performed. The test aimed to determine whether there was statistical evidence that the mean difference between paired observations was significantly different from zero. We report Cohen's *d*, a measure of effect size, to provide additional information about the magnitude of the difference in means to aid in interpreting the practical significance of the observed difference. It is computed as

$$d = \frac{M_1 - M_2}{SD_{pooled}} \tag{9}$$

where the numerator is the difference between the means of the two measures M_1 and M_2 and the denominator SD_{pooled} is the pooled standard deviations, a weighted average of the standard deviation of the two groups [69]. The effect sizes were interpreted as very small, small, medium, large, very large, or huge [70], as specified in Table 7. When the Shapiro–Wilk Test revealed that the normality assumption for the differences between the two groups of Index variant values was not met, the Wilcoxon Signed-Rank Test, a non-parametric equivalent to the Paired-Samples *t*-Test, was used instead. In such

cases, the Matched-Pairs Rank-Biserial Correlation (r_c) was used to measure the effect size, calculated as the difference between two proportions as

$$r_c = \frac{N_{RP}}{N_R} - \frac{N_{RN}}{N_R} \tag{10}$$

where N_{RP} signifies the number of positive ranks, N_{RN} the number of negative ranks, and N_R the total number of ranks. In proportion to the effect size, the coefficient's value could range from -1 to 1. A coefficient with a value of 0 indicates that there is no effect, i.e., no difference in the data between groups. Positive values imply that the data in the reference group tend to be more than the data in the comparison group. In contrast, negative values indicate that the data in the reference group tend to be fewer than in the comparison group [71].

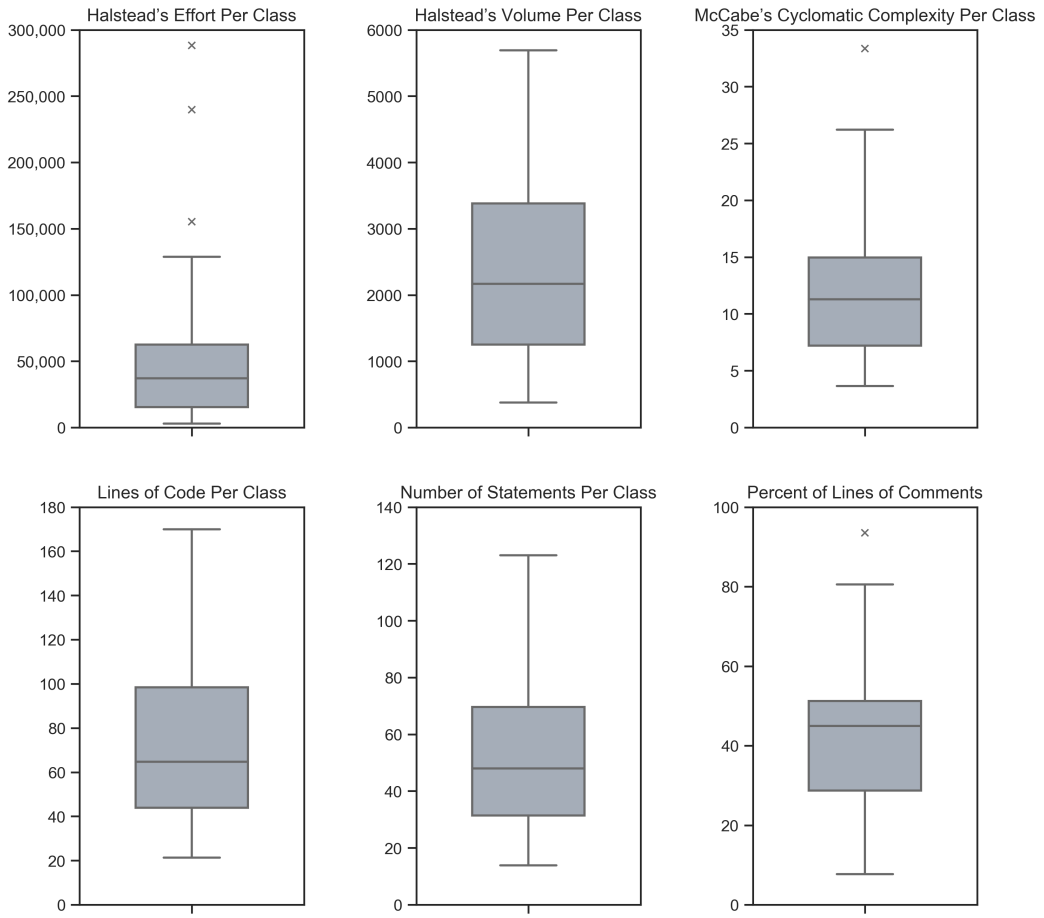

Figure 1. Boxplots of software metric values averaged per class.

Table 7. Interpretation of Cohen's *d* effect size.

Cohen's *d* Effect Size	Interpretation of the Effect Magnitude
$0.01 \leq \lvert d \rvert < 0.2$	Very small effect
$0.2 \leq \lvert d \rvert < 0.5$	Small effect
$0.5 \leq \lvert d \rvert < 0.8$	Medium effect
$0.8 \leq \lvert d \rvert < 1.2$	Large effect
$1.2 \leq \lvert d \rvert < 2$	Very large effect
$2 \leq \lvert d \rvert$	Huge effect

To answer RQ2.1, a pairwise correlation analysis was carried out between pairs of Index variants to examine the association between variant values. Based on the Shapiro–Wilk

Test, a parametric correlation coefficient was utilized, the Pearson Correlation Coefficient (r). It is an indicator of the strength and direction of a linear relationship between two variables. It has a possible range of values between -1 and 1, with -1 denoting a perfect negative linear relationship, 0 no relationship, and 1 a perfect positive linear relationship. According to the coefficient values described in Table 8, the strength of a correlation coefficient was interpreted as negligible, weak, moderate, strong, or very strong [72]. Following that, multidimensional scaling was conducted. The algorithm assisted in obtaining quantitative estimations of similarity and dissimilarity across Index variants in a low-dimensional vector space, i.e., a two-dimensional space in our case, enabling a graphical representation of the underlying relational structures contained therein. The algorithm conveyed the (dis)similarities between the set of variants in a lower-dimensional representation of the data in a way that preserved the relative (dis)similarities between the variants, and the distances respected well the distance in the original high-dimensional space. Accordingly, similar Index variants were spatially located closer together, and dissimilar variants were located proportionately further apart [73]. We performed metric multidimensional scaling using a distance matrix built on pairwise dissimilarity scores between Index variants, based on the previously conducted correlation analysis. The distance matrix, in our case a 7×7 symmetric square matrix, determined how dissimilar all the pairs of Index variants were from each other, based on the dissimilarity score, computed as

$$DS_{a,b} = 1 - |r_{a,b}| \tag{11}$$

where $DS_{a,b}$ is the dissimilarity score between Index variants a and b and $|r_{a,b}|$ is the absolute Pearson Correlation Coefficient of the two variants. It should be noted that the distance matrix's principal diagonal was equal to zero because the dissimilarity score between the pair of the same Index variant was always zero. Finally, hierarchical agglomerative clustering was performed to identify clusters, i.e., groups of similar Index variants, based on the computed pairwise dissimilarities scores in a way that resulted in maximum intra-cluster similarity and maximum inter-cluster dissimilarity. Each Index variant began as a single individual cluster. Then, the algorithm merged the two most-similar clusters based on the distance metric iteratively. We used the complete linkage method to calculate the distances, which defines the distance between two clusters as the maximum distance between any two individuals in the two clusters. Formally, the method is defined as

$$D(A, B) = max(dist(a, b));\ a \in A \wedge b \in B \tag{12}$$

where $D(A, B)$ represents the distance between clusters A and B and $dist(a, b)$ represents the distance between all individuals a in cluster A and individuals b in B [74].

Table 8. Interpretation of Pearson's and Kendall's Correlation Coefficient.

Pearson's Correlation	Kendall's τ-b Correlation	Interpretation of the Correlation
$0.0 \leq \|r\| < 0.1$	$0.0 \leq \|\tau\| < 0.1$	Negligible correlation
$0.1 \leq \|r\| < 0.4$	$0.1 \leq \|\tau\| < 0.4$	Weak correlation
$0.4 \leq \|r\| < 0.7$	$0.4 \leq \|\tau\| < 0.7$	Moderate correlation
$0.7 \leq \|r\| < 0.9$	$0.7 \leq \|\tau\| < 0.9$	Strong correlation
$0.9 \leq \|r\| \leq 1$	$0.9 \leq \|\tau\| \leq 1$	Very strong correlation

To answer RQ2.2, the link between Index variants was analyzed in the context of comparing maintainability with versions of the same software system using correlation analysis for each case study. Because we had a moderate sample size ($n > 50$), we employed the Kolmogorov–Smirnov Test to check the assumptions of normality. Based on the results, a non-parametric Kendall's τ-b Correlation Coefficient was used. Kendall's Correlation Coefficient, like Pearson's, is used to assess the degree of association between two variables, in our case two Index variants. The strength of correlation was interpreted as defined in Table 8. Next, multidimensional scaling with correlation-based distances was used to depict

similarities and dissimilarities for each case study, as in RQ2.1. In this case, Kendall's τ-b Correlation Coefficients were used to construct the distance matrix for each case study. Next, to assess if the maintainability trends based on the measurements provided by the seven Index variants are comparable, we evaluated the data based on the semantic versioning of the releases, i.e., major.minor.patch, such as 3.23.1. First, trend analysis was performed to assess the consistency of Index variants' maintainability evolution indications from a more long-term perspective. We used the Mann–Kendall Trend Test, a non-parametric statistical test for determining the underlying monotonic trend, i.e., consistent upward or downward maintainability trend, throughout the evolution of a software system, and Sen's Slope (β) to estimate the magnitude of the trend [75]. Following that, we focused on the short-term trends, i.e., changes in maintainability from a previously released version of a software system, calculated as

$$\Delta MI_i(j-1, j) = MI_{i,j} - MI_{i,j-1} \tag{13}$$

where $\Delta MI_i(j-1, j)$ is the change in the Index variant MI_i, j is the current release of a software system, and $j-1$ is the preceding release.

5. Results and Data Analysis

5.1. RQ1: Maintainability Measurement in a Software System

For each software system under study, values of the Index variants are visually presented in Figure 2. We can observe that, for each subject software, the single-valued maintainability evaluations by various Index variants were not the same. Nevertheless, certain general patterns in the Index values can be identified. The values of the Index variants that do not consider measuring code documentation, i.e., Index variants without the comment part, are on the left side of the plot, whereas the Index variants that take code comments into account in their computations are on the right side. Thus, the values of the former tend to be lower than the latter. Although this was to some extent expected given how the equations of Index variants with the comment part are formulated, an interesting observation is that the gap between the two groups of Index variants appears to be somewhat consistent for all software systems. Furthermore, we can observe that the gap tends to be slightly smaller when there is little code documentation, i.e., when the percentage of comment lines is low (e.g., in the case of *S19*, *S44*, *S45*), and the gap tends to be larger when the percentage of comment lines is higher (e.g., in the case of *S30*, *S31*). A closer examination of the individuals of the two groups reveals there is more variation in the specific Index variant values. The two lowest values are reported by $MI_{impr.}$ and MI_{LOC}. In a large majority of subject software (88.9%), MI_{LOC} has the lowest value, while in 11.1% of software, the lowest value is reported by $MI_{impr.}$. For the second-lowest value, the situation is reversed. Next, the third- and the fourth-lowest values are reported by either MI_{JH} or $MI_{orig.}$. In 60% of software systems, the third-lowest value is reported by MI_{JH} and in 40% by $MI_{orig.}$. The situation is reversed with the fourth-lowest value. The next value in the rankings, i.e., the third-highest value, is, in all cases, $MI_{impr.(CM)}$. $MI_{JH(CM)}$ and $MI_{orig.(CM)}$ report the two highest values. The highest value is reported by $MI_{orig.(CM)}$ in 60% of subject software and $MI_{JH(CM)}$ in 40%. For the second-highest values, 60% are reported by $MI_{JH(CM)}$ and 40% by $MI_{orig.(CM)}$. In Table 9, descriptive statistics about the Index variant values are provided, supporting the previously stated observations. In general, $MI_{impr.}$ and MI_{LOC} suggest lower maintainability for the same software system than MI_{JH} or $MI_{orig.}$, as the values of MI_{JH} and $MI_{orig.}$ tend to be higher than those of $MI_{impr.}$ and MI_{LOC}. Additionally, $MI_{JH(CM)}$ and $MI_{orig.(CM)}$ suggest higher maintainability in comparison to $MI_{impr.(CM)}$, since the value of $MI_{impr.(CM)}$ tends to be lower than that of $MI_{JH(CM)}$ and $MI_{orig.(CM)}$.

Figure 2. Dot plot of Index variant values for each subject software system.

Table 9. Descriptive statistics and normality test of Index variant values.

	$MI_{orig.}$	$MI_{orig.(CM)}$	$MI_{impr.}$	$MI_{impr.(CM)}$	MI_{LOC}	MI_{JH}	$MI_{JH(CM)}$
Avg.	66.66	112.41	60.90	106.64	59.23	66.37	112.11
95% CI	[62.64, 70.69]	[108.35, 116.47]	[56.73, 65.07]	[102.48, 110.8]	[54.87, 63.59]	[62.13, 70.61]	[107.87, 116.36]
Med.	67.74	111.37	61.04	105	59.98	66.74	112.15
Std.	13.39	13.51	13.88	13.84	14.51	14.13	14.14
Min.	44.29	82.70	36.90	78.44	33.45	42.54	83.70
Max.	94.74	136.82	89.78	131.86	88.23	96.71	138.78
Skew.	0.06	0.03	0.06	0.04	−0.01	0.10	0.08
Kurt.	−1.00	−0.69	−0.93	−0.72	−0.97	−0.90	−0.68
SW	W = 0.96, p = 0.147	W = 0.98, p = 0.461	W = 0.97, p = 0.207	W = 0.97, p = 0.389	W = 0.97, p = 0.256	W = 0.96, p = 0.165	W = 0.97, p = 0.352

Avg. = average, CI = confidence interval, Med. = median, Std. = standard deviation, Min. = minimum, Max. = maximum, Skew = skewness, Kurt. = kurtosis, SW = Shapiro–Wilk Test.

The boxplots in Figure 3 illustrate the pairwise differences between the values of different Index variants. The figure gives a general idea of the direction of the differences, i.e., whether they are generally positive, negative, or inconclusive. This helps show which Index variant has higher values between the two variants. For example, a consistent positive difference is observed between $MI_{orig.(CM)}$ and $MI_{orig.}$, while the results between $MI_{orig.}$

and MI_{JH} are inconsistent, as some subject software systems have higher maintainability based on $MI_{orig.}$ (e.g., in the case of *S1*), while others have higher maintainability based on MI_{JH} (e.g., in the case of *S4*). The Index variants without comment parts have smaller differences from Index variants without comment parts compared to Index variants with comment parts. For instance, the mean value of the absolute difference between $MI_{orig.}$ and $MI_{impr.}$, MI_{LOC}, and MI_{JH} is 5.77, 7.43, and 1.50, respectively, while the mean value of the absolute difference between $MI_{orig.}$ and $MI_{orig.(CM)}$, $MI_{impr.(CM)}$, and $MI_{JH(CM)}$ is 45.74, 39.98, and 45.45, respectively. Similarly, it can be observed that the Index variants with comment parts have smaller differences from Index variants with comment parts compared to Index variants without comment parts. For instance, the mean value of the absolute difference between $MI_{orig.(CM)}$ and $MI_{impr.(CM)}$ and $MI_{JH(CM)}$ is 5.77 and 1.50, respectively, while the mean value of the absolute difference between $MI_{orig.(CM)}$ and $MI_{orig.}$, $MI_{impr.}$, MI_{LOC}, and MI_{JH} is 45.74, 51.51, 53.18, and 46.04, respectively.

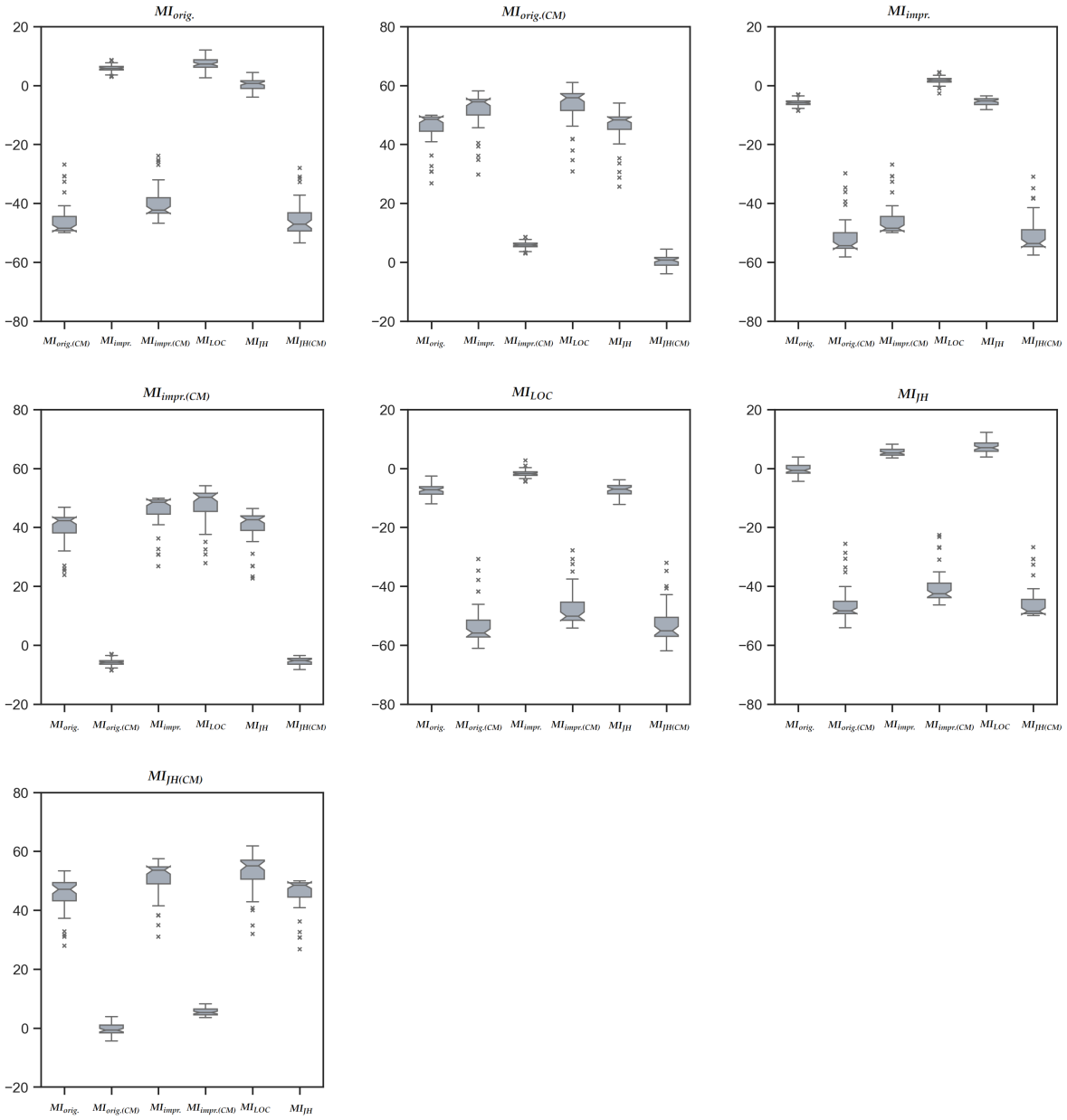

Figure 3. Boxplots of the pairwise differences between Index variant values.

The differences between the values of Index variants are evaluated in Table 10 using inferential statistics. Note that the Shapiro–Wilk Test was used to verify normality assumptions, and the appropriate test was then used, either the Paired Samples t-Test, when the normality assumptions were met, or the Wilcoxon Signed-Rank Test, when the normality assumptions were not met. To accompany the former, the average difference with the standard error and d effect size are reported in the table, while to accompany the latter, the median difference and r_c effect size are reported. All differences were found to be statistically significant, except for the differences between $MI_{orig.}$ and MI_{JH} and between $MI_{orig.(CM)}$ and $MI_{JH(CM)}$. In Figure 4, the boxplots of the Index variants with statistical significance annotations are depicted (only non-significant pairs are annotated). The boxplots of the non-significant pairs show a negligible difference, while the rest of the boxplots visibly differ. The d effect size ranges from huge (for $MI_{impr.} - MI_{orig.}$, $MI_{LOC} - MI_{orig.}$, $MI_{impr.(CM)} - MI_{orig.(CM)}$, $MI_{LOC} - MI_{JH}$), very large (for $MI_{LOC} - MI_{impr.}$), to very small (for $MI_{JH} - MI_{orig.}$, $MI_{JH(CM)} - MI_{orig.(CM)}$). For the last one, this means that the practical significance of the difference is very small. For all cases where the r_c effect size was used, its value equals -1 or 1. Where a positive effect size of 1 is reported, the Index variant values in the reference group are, in all cases, higher than the Index variant values in the comparison group (e.g., in the case of $MI_{orig.(CM)} - MI_{orig.}$). Contrariwise, when a negative effect size of -1 is reported, the Index variant values in the comparison group are, in all cases, higher than the Index variant values in the reference group (e.g., in the case of $MI_{orig.} - MI_{impr.(CM)}$).

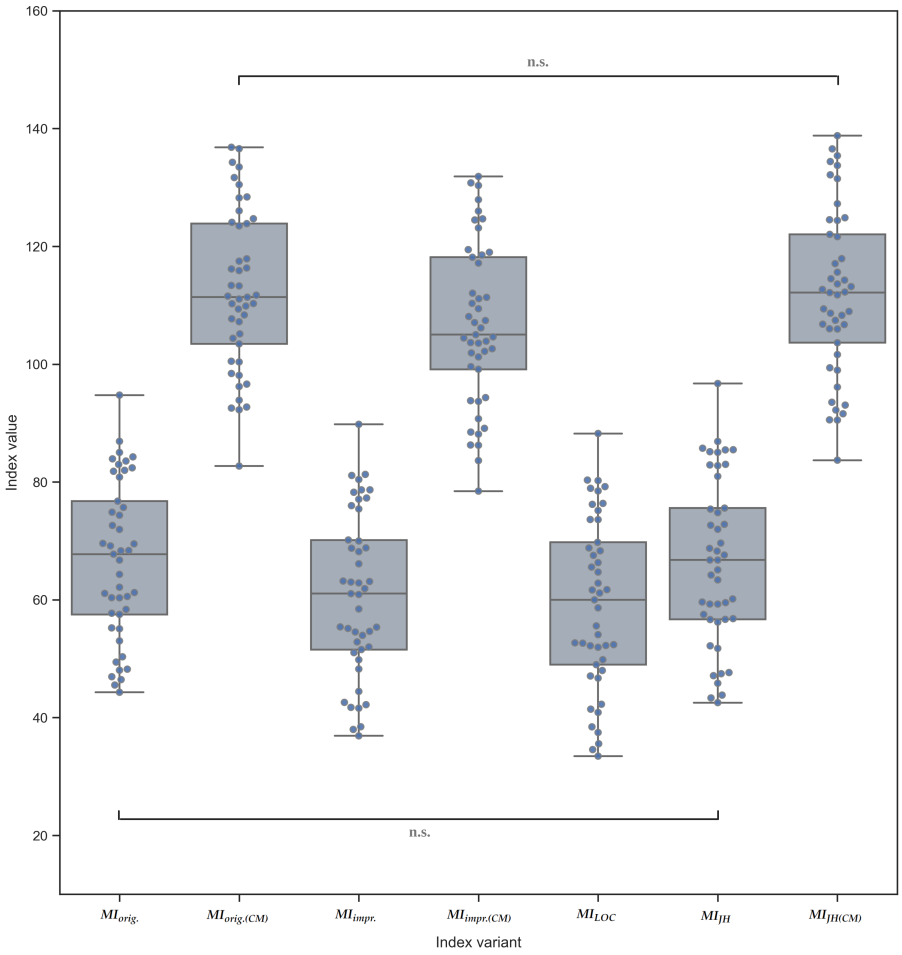

Figure 4. Boxplots of the Index variants values with statistical significance annotation (n.s. = not significant).

Table 10. Evaluation of the differences between the paired observations of Index variant values.

Comparison Group	Reference Group	Shapiro–Wilk Normality Test	Avg.(±SE [a]/Med. [b] Difference	Test t [a]/Z [b] Value	Statistical Significance	Effect Size d [a]/r_c [b]
$MI_{orig.}$	$MI_{orig.(CM)}$	W(45) = 0.71, $p < 0.001$ [b]	48.50	Z = −5.841	$p < 0.001$	$r_c = 1$
	$MI_{impr.}$	W(45) = 0.95, $p = 0.065$ [a]	−5.77 ± 0.19	t(44) = −30.54	$p < 0.001$	$d = -4.55$
	$MI_{impr.(CM)}$	W(45) = 0.81, $p < 0.001$ [b]	42.29	Z = −5.841	$p < 0.001$	$r_c = 1$
	MI_{LOC}	W(45) = 0.99, $p = 0.943$ [a]	−7.43 ± 0.31	t(44) = −23.88	$p < 0.001$	$d = -3.56$
	MI_{JH}	W(45) = 0.97, $p = 0.363$ [a]	−0.29 ± 0.27	t(44) = −1.09	$p = 0.280$	$d = -0.16$
	$MI_{JH.(CM)}$	W(45) = 0.87, $p < 0.001$ [b]	47.09	Z = −5.841	$p < 0.001$	$r_c = 1$
$MI_{orig.(CM)}$	MI_{orig}	W(45) = 0.71, $p < 0.001$ [b]	−48.50	Z = −5.841	$p < 0.001$	$r_c = -1$
	$MI_{impr.}$	W(45) = 0.74, $p < 0.001$ [b]	−54.40	Z = −5.841	$p < 0.001$	$r_c = -1$
	$MI_{impr.(CM)}$	W(45) = 0.95, $p = 0.065$ [a]	−5.77 ± 0.19	t(44) = −30.54	$p < 0.001$	d= −4.55
	MI_{LOC}	W(45) = 0.82, $p < 0.001$ [b]	−55.84	Z= −5.841	$p < 0.001$	r_c= −1
	MI_{JH}	W(45) = 0.80, $p < 0.001$ [b]	−48.34	Z = −5.841	$p < 0.001$	$r_c = -1$
	$MI_{JH.(CM)}$	W(45) = 0.97, $p = 0.363$ [a]	−0.29 ± 0.27	t(44) = −1.09	$p = 0.280$	$d = -0.16$
$MI_{impr.}$	MI_{orig}	W(45) = 0.95, $p = 0.065$ [a]	5.77 ± 0.19	t(44) = 30.54	$p < 0.001$	$d = 4.55$
	$MI_{orig.(CM)}$	W(45) = 0.74, $p < 0.001$ [b]	54.40	Z = −5.841	$p < 0.001$	$r_c = 1$
	$MI_{impr.(CM)}$	W(45) = 0.71, $p < 0.001$ [b]	48.50	Z = −5.841	$p < 0.001$	$r_c = 1$
	MI_{LOC}	W(45) = 0.95, $p = 0.077$ [a]	−1.67 ± 0.20	t(44) = −8.42	$p < 0.001$	$d = -1.26$
	MI_{JH}	W(45) = 0.94, $p = 0.023$ [b]	5.26	Z = −5.841	$p < 0.001$	$r_c = 1$
	$MI_{JH.(CM)}$	W(45) = 0.79, $p < 0.001$ [b]	53.6	Z = −5.841	$p < 0.001$	$r_c = 1$
$MI_{impr.(CM)}$	MI_{orig}	W(45) = 0.81, $p < 0.001$ [b]	−42.29	Z = −5.841	$p < 0.001$	r_c= −1
	$MI_{orig.(CM)}$	W(45) = 0.95, $p = 0.065$ [a]	5.77 ± 0.19	t(44) = 30.54	$p < 0.001$	$d = 4.55$
	$MI_{impr.}$	W(45) = 0.71, $p < 0.001$ [b]	−48.50	Z = −5.841	$p < 0.001$	$r_c = -1$
	MI_{LOC}	W(45) = 0.79, $p < 0.001$ [b]	−50.20	Z = −5.841	$p < 0.001$	$r_c = -1$
	MI_{JH}	W(45) = 0.78, $p < 0.001$ [b]	−42.58	Z = −5.841	$p < 0.001$	$r_c = -1$
	$MI_{JH.(CM)}$	W(45) = 0.94, $p = 0.023$ [b]	5.26	Z = −5.841	$p < 0.001$	$r_c = 1$
MI_{LOC}	MI_{orig}	W(45) = 0.99, $p = 0.943$ [a]	7.43 ± 0.31	t(44) = 23.88	$p < 0.001$	$d = 3.56$
	$MI_{orig.(CM)}$	W(45) = 0.82, $p < 0.001$ [b]	55.84	Z = −5.841	$p < 0.001$	$r_c = 1$
	$MI_{impr.}$	W(45) = 0.95, $p = 0.077$ [a]	1.67 ± 0.20	t(44) = 8.42	$p < 0.001$	$d = 1.26$
	$MI_{impr.(CM)}$	W(45) = 0.79, $p < 0.001$ [b]	50.20	Z = −5.841	$p < 0.001$	$r_c = 1$
	MI_{JH}	W(45) = 0.98, $p = 0.576$ [a]	7.14 ± 0.29	t(44) = 24.48	$p < 0.001$	$d = 3.65$
	$MI_{JH.(CM)}$	W(45) = 0.85, $p < 0.001$ [b]	55.11	Z = −5.841	$p < 0.001$	$r_c = 1$
MI_{JH}	MI_{orig}	W(45) = 0.97, $p = 0.363$ [a]	0.29 ± 0.27	t(44) = 1.09	$p = 0.280$	$d = 0.16$
	$MI_{orig.(CM)}$	W(45) = 0.80, $p < 0.001$ [b]	48.34	Z = −5.841	$p < 0.001$	$r_c = 1$
	$MI_{impr.}$	W(45) = 0.94, $p = 0.023$ [b]	−5.26	Z = −5.841	$p < 0.001$	$r_c = -1$
	$MI_{impr.(CM)}$	W(45) = 0.78, $p < 0.001$ [b]	42.58	Z = −5.841	$p < 0.001$	$r_c = 1$
	MI_{LOC}	W(45) = 0.98, $p = 0.576$ [a]	−7.14 ± 0.29	t(44) = −24.48	$p < 0.001$	$d = -3.65$
	$MI_{JH.(CM)}$	W(45) = 0.71, $p < 0.001$ [b]	48.50	Z = −5.841	$p < 0.001$	$r_c = 1$
$MI_{JH(CM)}$	MI_{orig}	W(45) = 0.87, $p < 0.001$ [b]	−47.09	Z = −5.841	$p < 0.001$	$r_c = -1$
	$MI_{orig.(CM)}$	W(45) = 0.97, $p = 0.363$ [a]	0.29 ± 0.27	t(44) = 1.09	$p = 0.280$	$d = 0.16$
	$MI_{impr.}$	W(45) = 0.79, $p < 0.001$ [b]	−53.61	Z = −5.841	$p < 0.001$	$r_c = -1$
	$MI_{impr.(CM)}$	W(45) = 0.94, $p = 0.023$ [b]	−5.26	Z = −5.841	$p < 0.001$	$r_c = -1$
	MI_{LOC}	W(45) = 0.85, $p < 0.001$ [b]	−55.11	Z = −5.841	$p < 0.001$	$r_c = -1$
	MI_{JH}	W(45) = 0.71, $p < 0.001$ [b]	−48.50	Z= −5.841	$p < 0.001$	r_c= −1

[a] Paired Samples *t*-test, [b] Wilcoxon Signed-Rank Test, Avg. = average, SE = standard error, Med. = median.

5.2. RQ2.1: Maintainability Measurement in Different Software Systems

The scatter plot matrix representation with pairwise correlation analysis is shown in Figure 5. Based on the results of the normality tests reported in Table 9, the Pearson

Correlation Coefficient was used to analyze the relationships between the Index variants. The results showed that all Index variants are strongly or very strongly positively correlated. The most outliers, represented by blue dots that are furthest from the regression line, can be observed when assessing the relationship between an Index variant without and one with the comment part. That is why, for example, the correlation between MI_{LOC} and $MI_{JH(CM)}$ (r = 0.896) is lower compared to the correlation between MI_{LOC} and $MI_{impr.}$ (r = 0.997).

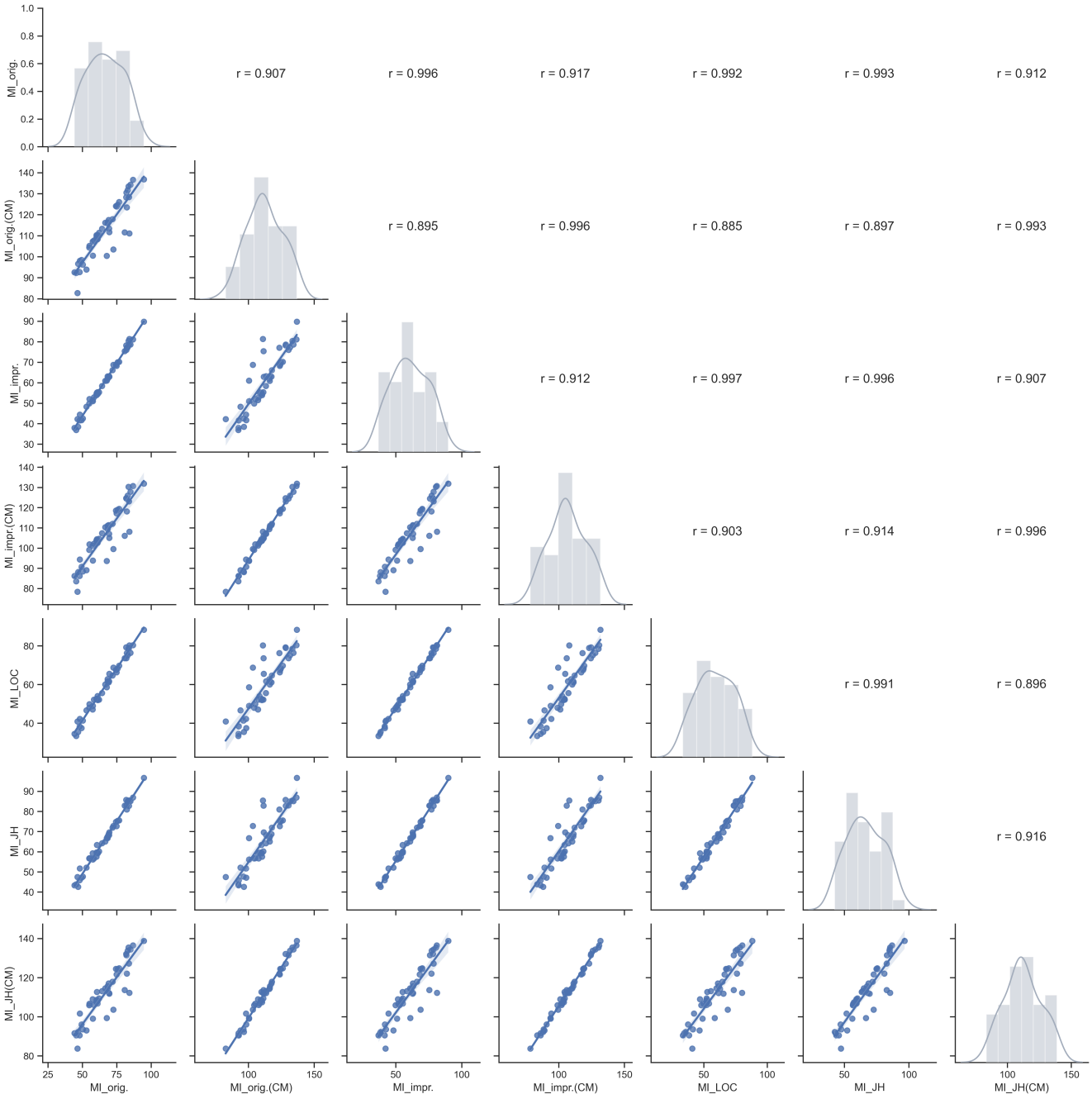

Figure 5. Scatter plot matrix representing pairwise correlation analyses (all reported correlations are significant ($p < 0.001$)) and variation in the data distribution.

To further analyze the multivariate relationships based on the Pearson Correlation Coefficient between the seven Index variants, multidimensional scaling was used to reduce the dimensions of the data to two. The resulting visualization seen in Figure 6 attempts to model the similarity and dissimilarity between the Index variants as the distances they

exhibit in the vector space. From the visualization, it can be observed that the Index variants with the comment part, namely $MI_{orig.(CM)}$, $MI_{impr.(CM)}$, and $MI_{JH(CM)}$, appear to be close together on the left side of the plot, while the Index variants without the comment part, namely $MI_{orig.}$, $MI_{impr.}$, MI_{LOC}, and MI_{JH}, appear close together on the right side of the plot. This observation indicates that Index variants without the comment part are more similar to each other in comparison to Index variants with the comment part. Similarly, this observation indicates that Index variants with the comment part are more similar to each other in comparison to Index variants without the comment part. The identified two groups are represented by the areas in the figure, which were manually annotated to emphasize the clear distinction between the two groups, as well as the close proximity of the Index variants within each group.

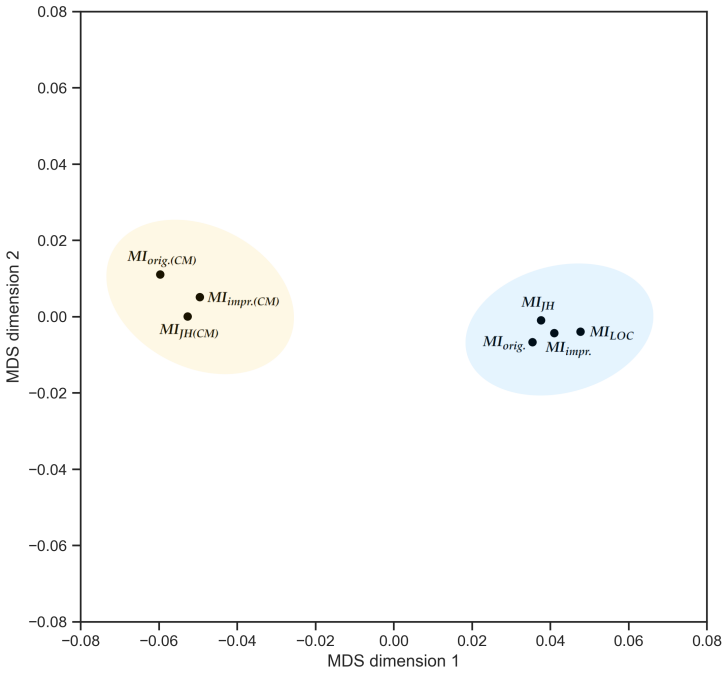

Figure 6. Visualizing similarities and dissimilarities after multidimensional scaling based on the Pearson Correlation Coefficient.

Instead of the manual identification of groups, to more accurately determine the similar groups of the Index variants, clustering on the Pearson Correlation Coefficients was performed. In Figure 7, a correlation heat map with a dendrogram that depicts the results of hierarchical clustering is presented. The dendrogram displays which Index variants are most similar to each other. The results showed that the first cluster created includes $MI_{impr.}$ and MI_{LOC}. The second cluster contains $MI_{impr.(CM)}$ and $MI_{JH(CM)}$, which is in the next step joined with $MI_{orig.(CM)}$. The fourth cluster includes $MI_{orig.}$ and MI_{JH}, which was in the next step combined with the first cluster of $MI_{impr.}$ and MI_{LOC}. Lastly, the two remaining clusters were merged into one. However, it can be noted that the merge threshold level of the two final clusters, as indicated by the distance in the dendrogram, is significantly higher compared to the previous clustering steps. These two distinct clusters, as identified by the hierarchical clustering, are the same as those that were manually identified earlier, i.e., the Index variants with the comment part and the Index variants without the comment part.

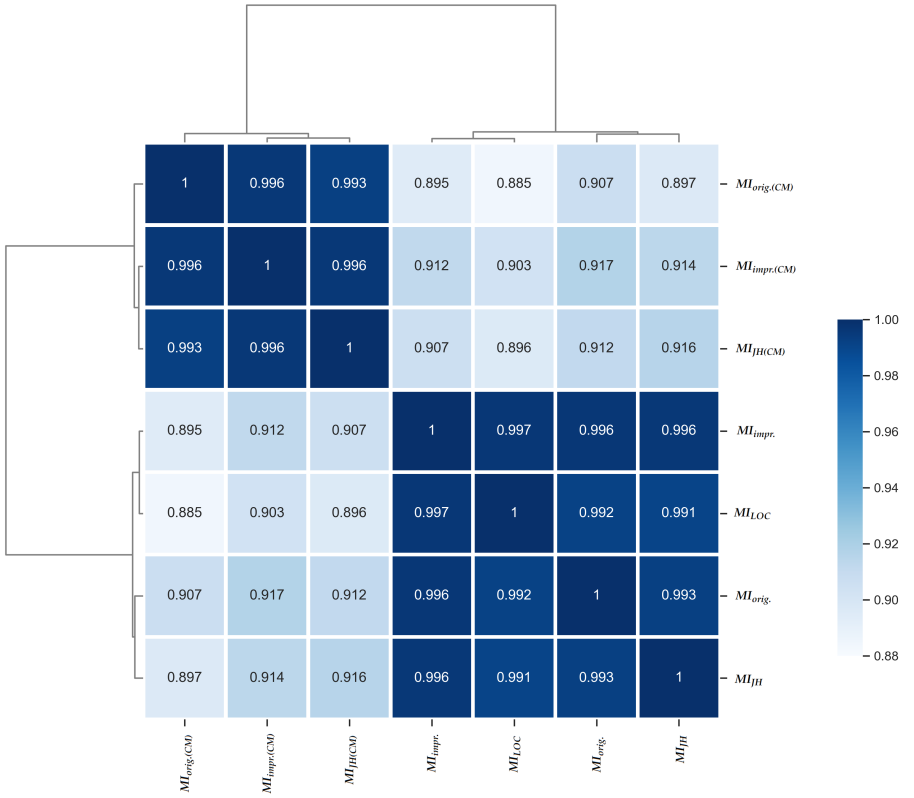

Figure 7. Correlation heat map and dendrogram of hierarchical agglomerative clustering.

5.3. RQ2.2: Maintainability Measurement in Versions of a Software System

To assess the relationship between Index variants in the context of comparing the maintainability of versions of the same software system, a correlation analysis was conducted for each case study. First, in Table 11, the results of the normality tests are reported. Even when a parametric correlation coefficient may have been appropriate to use (e.g., in the case of assessing the relationship between $MI_{orig.}$ and $MI_{orig.(CM)}$ for *CS2*), we opted to assess all relationships using Kendall's τ-b Correlation Coefficient to ensure consistency in comparing and analyzing the correlations across all pairs. The results of the correlation analysis for each case study are presented in Table 12. The correlation coefficients are positive, but tend to be weaker than those reported in RQ2.1, particularly for *CS1* and *CS2*. In the case of *CS1* and *CS2*, the correlation ranges from moderate to very strong, while in the case of *CS3*, the correlation is very strong across all pairs. Despite the pattern being less noticeable, the correlations between two Index variants with the comment part and between two Index variants without the comment part tend to be stronger in comparison to the correlation between two Index variants, one from each of the two groups. The case of *CS1* shows how three-metric Index variants can also be, in some cases, closely related to their four-metric alternative. For instance, the correlation between $MI_{orig.}$ and its version with the comment part, i.e., $MI_{orig.(CM)}$, is very strong (τ = 0.919), while it is only considered strong between $MI_{orig.}$ and MI_{LOC} (τ = 0.805), as well as between $MI_{orig.}$ and MI_{JH} (τ = 0.729).

The results of the correlation analysis were further analyzed through multidimensional scaling to gain a deeper understanding of the relationships between the Index variants. The resulting visualizations for each case study are presented in Figure 8. There is still a tendency for Index variants with comment parts to be close to each other in the vector space, as well as a tendency for Index variants without comment parts to be close to each other. However, the pattern is not as pronounced as in the previous analysis for RQ2.1. Herein, in some case studies, certain Index variants with the comment part are closer to certain Index variants without the comment part and vice versa. For example, in the case of *CS1*, $MI_{orig.}$ appears closer to $MI_{orig.(CM)}$ rather than to MI_{LOC} and MI_{JH}.

Table 11. Kolmogorov–Smirnov normality test.

Case Study	$MI_{orig.}$	$MI_{orig.(CM)}$	$MI_{impr.}$	$MI_{impr.(CM)}$	MI_{LOC}	MI_{JH}	$MI_{JH(CM)}$
CS1. AssertJ	W = 0.151, $p = 0.001$	W = 0.199, $p < 0.001$	W = 0.189, $p < 0.001$	W = 0.177, $p < 0.001$	W = 0.159, $p < 0.001$	W = 0.169, $p < 0.001$	W = 0.255, $p < 0.001$
CS2. Caffeine	W = 0.102, $p = 0.186$	W = 0.131, $p = 0.011$	W = 0.100, $p = 0.200$	W = 0.137, $p = 0.006$	W = 0.113, $p = 0.050$	W = 0.091, $p = 0.200$	W = 0.156, $p < 0.001$
CS3. Joda-Time	W = 0.404, $p < 0.001$	W = 0.395, $p < 0.001$	W = 0.404, $p < 0.001$	W = 0.399, $p < 0.001$	W = 0.404, $p < 0.001$	W = 0.404, $p < 0.001$	W = 0.402, $p < 0.001$

Table 12. Correlation analysis using Kendall's τ-b Correlation Coefficient (all reported correlations are significant ($p < 0.001$)).

***CS1.* AssertJ**							
	$MI_{orig.}$	$MI_{orig.(CM)}$	$MI_{impr.}$	$MI_{impr.(CM)}$	MI_{LOC}	MI_{JH}	$MI_{JH(CM)}$
$MI_{orig.}$	1	0.919	0.942	0.870	0.805	0.729	0.724
$MI_{orig.(CM)}$	0.919	1	0.873	0.943	0.745	0.663	0.789
$MI_{impr.}$	0.942	0.873	1	0.838	0.860	0.780	0.750
$MI_{impr.(CM)}$	0.870	0.943	0.838	1	0.716	0.630	0.813
MI_{LOC}	0.805	0.745	0.860	0.716	1	0.887	0.683
MI_{JH}	0.729	0.663	0.780	0.630	0.887	1	0.667
$MI_{JH(CM)}$	0.724	0.789	0.750	0.813	0.683	0.667	1
***CS2.* Caffeine**							
	$MI_{orig.}$	$MI_{orig.(CM)}$	$MI_{impr.}$	$MI_{impr.(CM)}$	MI_{LOC}	MI_{JH}	$MI_{JH(CM)}$
$MI_{orig.}$	1	0.629	0.909	0.651	0.892	0.885	0.629
$MI_{orig.(CM)}$	0.629	1	0.679	0.972	0.703	0.659	0.945
$MI_{impr.}$	0.909	0.679	1	0.700	0.948	0.864	0.663
$MI_{impr.(CM)}$	0.651	0.972	0.700	1	0.724	0.679	0.923
MI_{LOC}	0.892	0.703	0.948	0.724	1	0.864	0.685
MI_{JH}	0.885	0.659	0.864	0.679	0.864	1	0.663
$MI_{JH(CM)}$	0.629	0.945	0.663	0.923	0.685	0.663	1
***CS3.* Joda-Time**							
	$MI_{orig.}$	$MI_{orig.(CM)}$	$MI_{impr.}$	$MI_{impr.(CM)}$	MI_{LOC}	MI_{JH}	$MI_{JH(CM)}$
$MI_{orig.}$	1	0.957	0.988	0.963	0.977	0.988	0.983
$MI_{orig.(CM)}$	0.957	1	0.945	0.994	0.934	0.945	0.971
$MI_{impr.}$	0.988	0.945	1	0.951	0.984	0.988	0.971
$MI_{impr.(CM)}$	0.963	0.994	0.951	1	0.940	0.951	0.974
MI_{LOC}	0.977	0.934	0.984	0.940	1	0.980	0.963
MI_{JH}	0.988	0.945	0.988	0.951	0.980	1	0.974
$MI_{JH(CM)}$	0.983	0.971	0.971	0.974	0.963	0.974	1

The consistency of the long-term trends in maintainability for each case study was analyzed using Mann–Kendall Trend Tests, and the results are presented in Table 13. The results suggest that the long-term monotonic trends in maintainability are all statistically significant and point in the same direction; the maintainability trends are negative, indicating that maintainability is consistently decreasing over time. Nonetheless, the choice of Index variant affects the magnitude of the trend, with different variants yielding different results. In the case of *CS2*, it can be observed that the magnitude of the trends is generally slightly higher when maintainability is assessed using Index variants with comment parts ($\beta = -0.051$, $\beta = -0.054$, and $\beta = -0.056$ for $MI_{orig.(CM)}$, $MI_{impr.(CM)}$, and $MI_{JH(CM)}$, respectively) compared to when maintainability is assessed using Index variants without comment parts ($\beta = -0.010$, $\beta = -0.011$, $\beta = -0.014$, and $\beta = -0.015$ for $MI_{orig.}$, $MI_{impr.}$,

MI_{JH}, and MI_{LOC}, respectively). Additionally, a pattern can be noted between the magnitudes of the trends based on the maintainability assessment made by a three-metric Index variant and the one made by its four-metric alternative across all three case studies. For instance, in the case of *CS1*, the magnitude of the trend assessed by $MI_{orig.(CM)}$ ($\beta = -0.064$) is higher than the one by $MI_{orig.}$ ($\beta = -0.056$); the magnitude of the trend assessed by $MI_{impr.(CM)}$ ($\beta = -0.055$) is higher than the one by $MI_{impr.}$ ($\beta = -0.048$); the magnitude of the trend assessed by $MI_{JH(CM)}$ ($\beta = -0.033$) is higher than the one by MI_{JH} ($\beta = -0.025$).

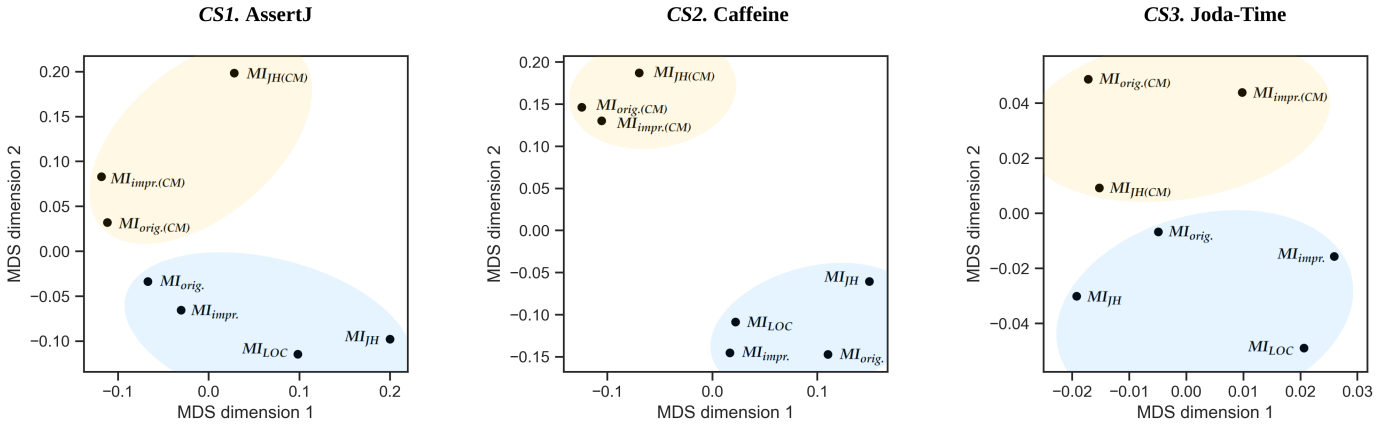

Figure 8. Visualizing similarities and dissimilarities after multidimensional scaling based on Kendall's τ-b Correlation Coefficient.

Table 13. Mann–Kendall Trend analyses.

Case Study	$MI_{orig.}$	$MI_{orig.(CM)}$	$MI_{impr.}$	$MI_{impr.(CM)}$	MI_{LOC}	MI_{JH}	$MI_{JH(CM)}$
CS1. AssertJ	$Z = -7.640$, $\beta = -0.056$, $p < 0.001$	$Z = -8.316$, $\beta = -0.064$, $p < 0.001$	$Z = -7.153$, $\beta = -0.048$, $p < 0.001$	$Z = -8.672$, $\beta = -0.055$, $p < 0.001$	$Z = -5.718$, $\beta = -0.031$, $p < 0.001$	$Z = -4.650$, $\beta = -0.025$, $p < 0.001$	$Z = -7.841$, $\beta = -0.033$, $p < 0.001$
CS2. Caffeine	$Z = -3.485$, $\beta = -0.010$, $p < 0.001$	$Z = -6.783$, $\beta = -0.051$, $p < 0.001$	$Z = -3.871$, $\beta = -0.011$, $p < 0.001$	$Z = -6.509$, $\beta = -0.054$, $p < 0.001$	$Z = -4.120$, $\beta = -0.014$, $p < 0.001$	$Z = -4.369$, $\beta = -0.015$, $p < 0.001$	$Z = -7.156$, $\beta = -0.056$, $p < 0.001$
CS3. Joda-Time	$Z = -9.302$, $\beta = -0.030$, $p < 0.001$	$Z = -9.276$, $\beta = -0.032$, $p < 0.001$	$Z = -9.302$, $\beta = -0.032$, $p < 0.001$	$Z = -9.276$, $\beta = -0.034$, $p < 0.001$	$Z = -9.234$, $\beta = -0.037$, $p < 0.001$	$Z = -9.334$, $\beta = -0.024$, $p < 0.001$	$Z = -9.324$, $\beta = -0.027$, $p < 0.001$

To examine the short-term trends in the maintainability of each case study, the differences in maintainability from one software release to its previous version are presented separately for each Index variant under study in Figure 9. Since the differences in maintainability have a large range of values, the Y axis uses a symmetrical logarithmic scale to help the reader better observe all these changes. This scale represents small values, i.e., small differences in maintainability, on a linear scale and larger values, i.e., large differences, on a logarithmic scale. Blue highlights are placed behind the lines to emphasize changes that majorly differ from those observed in the maintainability assessments made using other Index variants, or at least from most of them. In most cases, it can be noted that the changes are consistent; if, according to one Index variant, a positive change is shown, it is likely that this change is also regarded as positive according to the assessments made using other Index variants. When differences in the direction of the maintainability change between two consecutive releases do occur, the directions of the change are mostly consistent between Index variants with and those without the comment part. However, while the directions of the changes are, in most cases, aligned, the magnitudes of these changes often vary for assessments made by all Index variants.

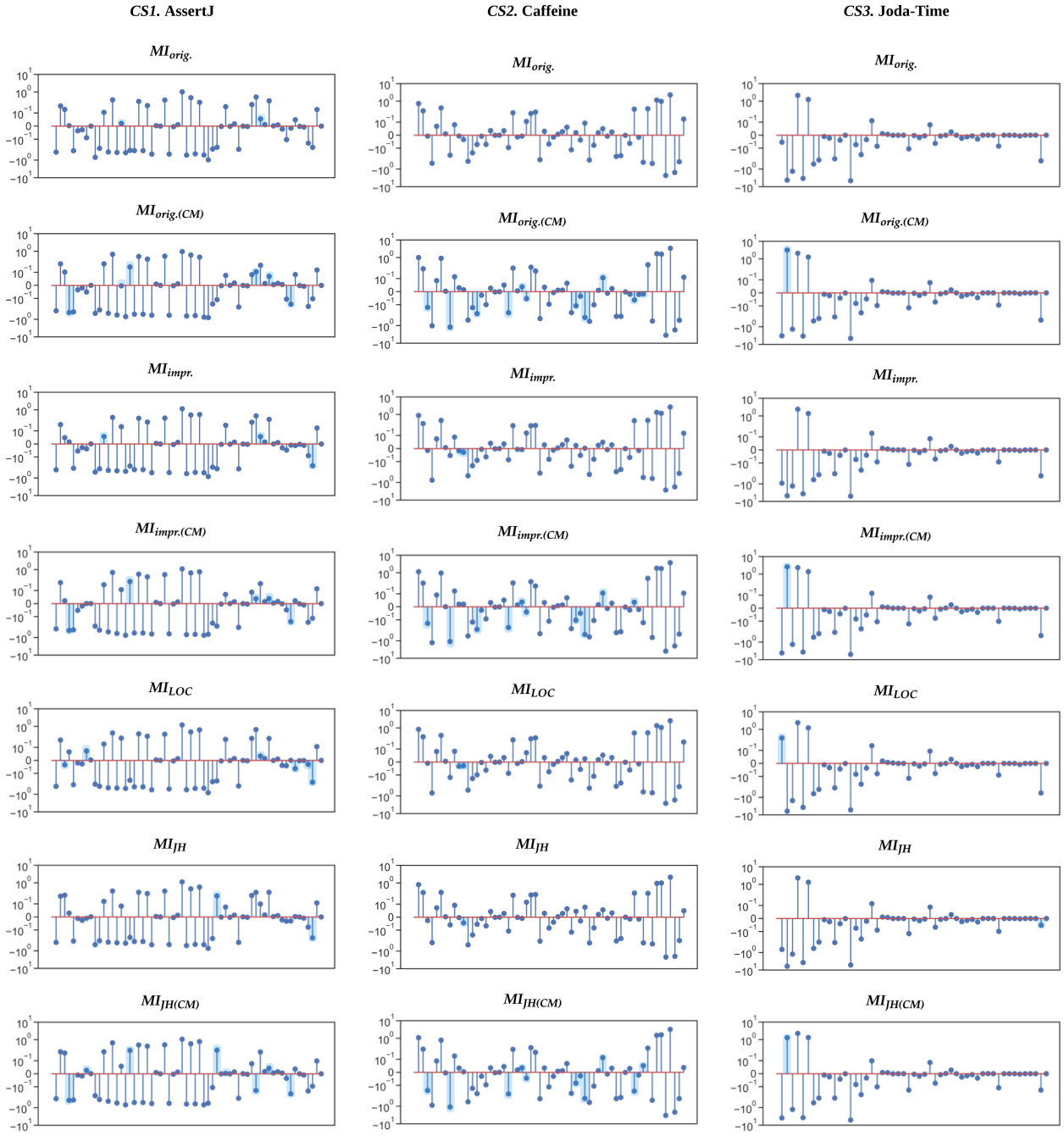

Figure 9. Lollipop plots depicting the change in Index variants' values from the previous release.

6. Discussion

6.1. Summary of Research Questions and Their Answers

6.1.1. RQ1: How Do Different Maintainability Index Variants Perform When Utilized for the Maintainability Measurement of a Single Object-Oriented Software System?

The results of different variants of an Index used to assess the maintainability of object-oriented software systems can vary greatly. To ensure the most-accurate assessment, the specific variant selected should take into account the relevant aspects of maintainability that best represent the maintainability of the software system in question. For example, if code documentation significantly benefits software maintainability, a variant incorporating code comments into the Index computation should be used. Note that some human analysis of the comments is recommended to aid this decision [10,36]. The differences between the values produced by different Index variants are, in most cases, significant

and should not be overlooked. Typically, the Index variants without the comment part have closer values, as well as the Index variants with the comment part. Additionally, the former tend to have lower values compared to the latter. As a result, this makes it difficult to compare values obtained from different variants directly. Furthermore, using originally proposed thresholds for conventional systems to convert the maintainability assessment on an interval scale, which ranges from an unbounded negative number to 171, into categorical, i.e., low, medium, and high maintainability, may not be appropriate without considering the object-oriented nature of the software system and the specific Index variant used to make the maintainability assessment.

6.1.2. RQ2.1: How Do Different Maintainability Index Variants Perform When Utilized for Maintainability Measurement between Different Software Systems?

Although there are differences in the values produced by different variants of an Index used to measure software system maintainability, the variants tend to behave similarly when comparing maintainability between systems. The relationship between all pairs of Index variants under study is positive and strong to very strong. When comparing two software systems based on their maintainability assessments from different Index variants, if one software is easier to maintain than the other according to one variant, it is very likely that the same holds true according to any other variant. Moreover, when comparing the maintainability of software systems using different variants of the Index, very similar outputs can be expected for all variants without the comment part and all variants with the comment part. On the other hand, when comparing the maintainability of systems using two Index variants, one from each of the two groups, the consistency in the maintainability comparison is expected to be lower. These findings reinforce the importance of carefully considering which variant of the Index to use for the maintainability assessment, namely taking into account whether or not code documentation should be considered.

6.1.3. RQ2.2: How Do Different Maintainability Index Variants Perform When Utilized for Maintainability Measurement between Versions of the Same Software System?

When comparing the maintainability of different versions of the same software using different Index variants, the relationship between the variants remains positive, but can range from moderate to very strong. It can be observed that, in the case when comparing the maintainability between releases of the same software system, i.e., when changes in the source code are small, the Index variants tend to be more prone to non-consistency in the maintainability comparisons. This means that when focusing on finer-grained results posed by the Index, the selection of the Index variant has even more of an impact on the maintainability perception. Therefore, extra caution should be taken when using the Index variants for more specific assessments instead of general ones. Despite this, the Index variants without comment parts still tend to be more similar to each other than those without comment parts and vice versa. From a short-term perspective, the maintainability trends, i.e., changes in maintainability, are in the same direction, but can be of different strengths, while from a long-term perspective, the maintainability trends are in alignment, although the magnitude of the trends is not.

6.2. Theoretical and Practical Implications

Overall, the findings of this research have the potential to benefit both researchers and practitioners in the field of software engineering by providing a better understanding of how the selection of Index variant affects the perception of the maintainability of software systems.

One of the theoretical implications of this study that could be further exploited in software-engineering-related research is the insight into the characteristics of the evaluated variants when applied in different contexts, which was gained through the empirical evaluation and comparison of different variants of the Index. These findings contribute to the body of knowledge about software maintainability and are of interest to software researchers. Researchers should carefully examine which Index variants should be used

in their studies, as the results can be impacted by the choices made. Furthermore, this study identified some research gaps and suggested some possible directions for future research efforts. For instance, Index variant thresholds for object-oriented software should be refined in the future while also considering the specifics of the Index variant used for maintainability assessment. For non-object-oriented software systems, an Index value below 65 indicates poor maintainability, between 65 and 85 indicates medium maintainability, and above 85 indicates high maintainability [10]. However, as Sjøberg et al. [19] already noted, these standard threshold values are inappropriate for object-oriented software since the classes tend to be generally smaller than in conventional software. Moreover, there is no clear scheme of Index thresholds for object-oriented systems. We add to this finding that the Index variant used for assessing maintainability should be taken into account when setting the thresholds. This would ensure that the threshold values appropriately reflect the software's maintainability characteristics.

In terms of practical implications, software practitioners can use the results of this research to guide their selection of the most-appropriate Index variant for assessing the maintainability of their software systems in an efficient and effective manner. By better understanding Index variants and the importance of the context in which an Index variant is used, practitioners can select the variant that best fits their specific needs and goals. Ignoring which Index variant is used to assess maintainability could lead to misleading perceptions. Additionally, by having a better understanding of the characteristics that contribute to maintainability and how they can be measured by different Index variants, practitioners can also make informed decisions about how to prioritize and focus their efforts to improve these aspects of their software during development, leading to improvements in software quality and the long-term sustainability of their software systems.

6.3. Threats to Validity

6.3.1. Internal Validity

The first threat to internal validity is related to the method used for data collection. All source codes for the included software systems were obtained from the original (i.e., not forked) GitHub repositories. To ensure all included software systems were following object-oriented principles, some manual inspection of the source code of subject software systems was performed by the authors. To reduce the likelihood of errors caused by humans, the process of collecting software metrics and computing the Index variants for all software under study was performed automatically by running scripts. The correctness and completeness of this process were manually verified on a sample of subjects. The JHawk tool was used to extract the source code metrics, as it is commonly used for metrics extraction of Java software. The Index variants' computation was performed according to the equations presented in Section 2.3.

The next threat to internal validity is related to the methods of data analysis used. Since the chosen analyses may bias the result, we explained the idea and motivation for each data analysis method used. When possible, the parameter settings of the data analysis method used were provided to the reader to give a better understanding of the analysis. For each hypothesis testing with statistical tests, we reviewed the underlying assumptions associated with each inferential statistic and reported the extent to which the assumptions were met.

6.3.2. External Validity

One major threat to external validity is related to the software systems on which the research is based, including sample size, selection bias, and subject characteristics. To address these threats, our research relied on a version of 45 different software systems and all versions of 3 software systems, a sample generally larger than that in related work. The selection was random, and our selected subjects represent a heterogeneous group of software systems in terms of software characteristics, including size, complexity, code documentation, maturity, and domain of use. However, the results could have been

different for a different group of software systems, as there is no guarantee that the selected software systems are good representatives. All systems studied were open-source and based on the Java programming language. Therefore, it is plausible that the results are different for closed-source software and may not be representative of the overall population, but only for open-source software. Furthermore, the results of the study may not generalize to the broader population of object-oriented systems based on programming languages other than Java. Even though the findings are likely valid for software based on other object-oriented languages, further research is needed to confirm them.

Another threat to external validity is the set of Index variants studied. We limited our selection to well-defined variants (i.e., variants with an adequate description of the metrics needed for computation, the process of the metrics' collection, and the process of Index calculation) that are commonly used in existing research on object-oriented software. Replication studies are needed to confirm and generalize the results to a more extensive set of Index variants.

7. Conclusions

Maintainability as a software characteristic is of paramount importance in software development because it substantially affects the long-term expenses associated with a software system. It is believed that a system that is difficult to maintain requires more resources to correct, improve, perfect, and adapt, which raises the overall cost of the software. Conversely, an easily maintainable system is believed to be more cost effective in carrying out such maintenance tasks while also reducing the software system's tendency towards deteriorating software quality. Therefore, it is essential to understand how maintainable a software system is as it is being developed.

Several aspects can impact the software's maintainability, including the code's complexity, the software's size, and the presence of documentation and comments that help describe the code. The Maintainability Index is frequently used as a single-valued measure for estimating the overall maintainability of software, taking these aspects into account in different ways through various Index variants. However, it is still unclear how different Index variants compare when applied to software systems. This paper evaluated and compared the Index variants by applying them to Java open-source software systems. The objective was to determine if and to what degree these variants are consistent with one another when used to assess the maintainability of a single software system and when used to compare the maintainability of different software systems.

The study's results showed that when comparing maintainability measurements from several Index variants, the perception of maintainability could be impacted by the choice of the Index variant used. Although different Index variants present different values when subjected to the same software system, the Index variant values are very strongly positively correlated and indicate in a similar manner how maintainability evolves between releases and from a more long-term perspective. Based on their maintainability measurement characteristics, the Index variants can be divided into two distinct clusters containing variants that are more similar to one another: Index variants without considering code comments and Index variants that include code comments in their computation. The differences in Index variants' values are the most notable when looking at the finer-grained results derived from different Index variants.

In the future, it would be interesting to extend our study to software systems written in other object-oriented programming languages, such as Python and C#. Another potential area of research is to conduct replication studies of existing work by analyzing the results from the perspective of using Index variants other than the one primarily selected. This may offer intriguing insights into whether and how the selection of Index variants affects the results and established findings in the software engineering research community. In addition, it would be valuable to examine the Index variants for maintainability assessments at levels other than the system level (i.e., packages, classes) since this study focused only on

examining Index variants at the overall system level. Future efforts could also be invested in investigating and determining the Index threshold levels for object-oriented software.

Author Contributions: Conceptualization, T.H. and B.Š.; methodology, T.H. and B.Š.; software, T.H.; validation, T.H. and B.Š.; formal analysis, T.H.; resources, B.Š.; investigation, T.H.; data curation, T.H.; writing—original draft preparation, T.H.; writing—review and editing, T.H. and B.Š.; visualization, T.H.; supervision, B.Š.; project administration, B.Š.; funding acquisition, B.Š. All authors have read and agreed to the published version of the manuscript.

Funding: The authors acknowledge the financial support from the Slovenian Research Agency (Research Core Funding No. P2-0057).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

RQ	Research question
$MI_{orig.}$	Original Maintainability Index
$MI_{orig.(CM)}$	Original Maintainability Index with comment part
$MI_{impr.}$	Improved Maintainability Index
$MI_{impr.(CM)}$	Improved Maintainability Index with comment part
MI_{LOC}	Maintainability Index using Lines Of Code only
MI_{JH}	JHawk Maintainability Index
$MI_{JH(CM)}$	JHawk Maintainability Index with comment part
MI_{VS}	Visual Studio Maintainability Index
$aveE$	Average Halstead's Effort per module
$aveV$	Average Halstead's Volume per module
$aveV(g)$	Average Cyclomatic Complexity per module
$aveLOC$	Average Lines of Code per module
$aveNOS$	Average Number of Statements per module
$perCM$	Average Percent of Lines of Comments per module
S	Software system
CS	Case study
$Avg.$	Average
$Med.$	Median
$Std.$	Standard deviation
$Min.$	Minimum
$Max.$	Maximum
$Skew.$	Skewness
$Kurt.$	Kurtosis
CI	Confidence interval
SE	Standard error
SW	Shapiro–Wilk Test
MDS	Multidimensional scaling

Appendix A. Maintainability Index Variant Values

Table A1. Index variant values (RQ1 and RQ2.1).

Software System	$MI_{orig.}$	$MI_{orig.(CM)}$	$MI_{impr.}$	$MI_{impr.(CM)}$	MI_{LOC}	MI_{JH}	$MI_{JH(CM)}$
S1. Activiti	68.36	115.88	61.90	109.41	62.82	66.74	114.26
S2. Angry IP Scanner	82.39	123.47	77.07	118.14	76.20	80.96	122.03
S3. Apache Ant	64.31	113.28	58.43	107.40	55.56	64.19	113.17
S4. Apache Commons Codec	48.21	98.09	44.44	94.32	42.26	51.74	101.63
S5. Apache Commons CSV	49.42	98.41	41.73	90.72	37.47	47.11	96.11
S6. Apache Commons DBCP	45.52	92.25	36.90	83.63	33.45	43.81	90.54
S7. Apache Commons Lang	60.33	109.36	54.63	103.67	52.68	59.61	108.64
S8. Apache HttpClient	69.14	117.49	63.00	111.34	61.14	68.73	117.07
S9. Apache PDFBox	57.51	107.20	51.55	101.23	47.06	59.26	108.95
S10. Apache POI	61.23	110.25	54.54	103.56	51.94	59.27	108.29
S11. Arduino	68.31	116.33	63.10	111.12	61.72	67.59	115.61
S12. Art of Illusion	46.44	82.70	42.19	78.44	40.87	47.44	83.70
S13. AssertJ	81.98	130.48	75.98	124.48	73.60	82.99	131.49
S14. Caffeine	69.49	113.37	63.18	107.06	61.66	68.26	112.15
S15. cglib	83.92	128.39	78.65	123.12	78.92	82.77	127.24
S16. DITA Open Toolkit	57.71	100.49	51.02	93.80	48.98	56.20	98.98
S17. EasyMock	76.74	126.03	70.14	119.44	69.75	75.56	124.85
S18. Ehcache	71.94	117.88	66.10	112.04	64.70	71.97	117.91
S19. FastJSON	84.26	111.06	81.28	108.08	80.23	85.44	112.24
S20. GeOxygene	58.36	107.70	52.84	102.18	52.19	56.67	106.01
S21. h2database	48.04	92.72	41.59	86.27	38.42	45.82	90.51
S22. Hibernate ORM	69.57	111.71	62.85	105.00	65.55	69.62	111.76
S23. iText7	55.22	105.14	51.99	101.91	49.83	56.80	106.72
S24. JabRef	67.74	100.38	61.04	93.67	58.63	66.74	99.38
S25. Jajuk	66.73	116.16	60.89	110.32	59.98	65.07	114.50
S26. JasperReports	60.56	109.87	55.13	104.44	52.63	63.37	112.68
S27. javaGeom	60.35	110.29	53.95	103.89	52.38	57.50	107.44
S28. Java Hamcrest	86.90	136.57	81.10	130.77	80.32	86.87	136.55
S29. Jenkins	75.68	124.69	69.98	118.99	68.30	75.39	124.40
S30. JFreeChart	50.32	96.20	42.57	88.46	41.42	47.64	93.53
S31. JFreeSVG	44.29	92.54	37.98	86.23	34.56	43.33	91.58
S32. JGraphT	55.07	104.39	49.81	99.13	47.99	56.64	105.96
S33. JMeter	62.13	111.37	55.39	104.63	52.24	60.14	109.38
S34. Joda-Time	46.93	96.61	38.44	88.11	35.55	42.54	92.21
S35. jsoup	53.01	93.88	48.24	89.10	46.70	52.18	93.04
S36. JUnit4	94.74	136.82	89.78	131.86	88.23	96.71	138.78
S37. JUnit5	82.97	131.68	77.28	125.99	75.15	85.02	133.73
S38. Mockito	81.81	128.23	78.25	124.67	79.20	85.72	132.14
S39. MPAndroidChart	61.08	108.35	55.34	102.62	54.08	59.51	106.78
S40. PowerMock	83.55	133.47	80.41	130.33	78.47	85.47	135.39
S41. SLF4J	74.87	123.85	68.17	117.15	66.31	72.65	121.64
S42. Spring	74.35	124.07	68.81	118.53	67.54	74.79	124.51
S43. Spring Boot	85.01	134.27	78.66	127.92	76.36	85.13	134.39
S44. TestNG	80.82	111.54	75.42	106.15	73.62	82.89	113.62
S45. YamlBeans	72.62	103.44	68.75	99.57	68.80	72.80	103.63

Table A2. Index variant values for *CS1.* AssertJ (RQ2.2).

Case Study	Release	$MI_{orig.}$	$MI_{orig.(CM)}$	$MI_{impr.}$	$MI_{impr.(CM)}$	MI_{LOC}	MI_{JH}	$MI_{JH(CM)}$
CS1. AsssertJ	1.0.0	85.29	134.60	78.95	128.26	75.67	84.71	134.02
CS1. AsssertJ	1.1.0	84.94	134.28	78.61	127.94	75.32	84.37	133.71
CS1. AsssertJ	1.2.0	85.10	134.47	78.75	128.13	75.49	84.55	133.92
CS1. AsssertJ	1.3.0	85.23	134.57	78.80	128.15	75.46	84.75	134.10
CS1. AsssertJ	1.4.0	85.23	134.16	78.82	127.75	75.52	84.78	133.72

Table A2. *Cont.*

Case Study	Release	$MI_{orig.}$	$MI_{orig.(CM)}$	$MI_{impr.}$	$MI_{impr.(CM)}$	MI_{LOC}	MI_{JH}	$MI_{JH(CM)}$
CS1. AsssertJ	1.5.0	84.94	133.79	78.54	127.39	75.24	84.50	133.35
CS1. AsssertJ	1.6.0	84.90	133.75	78.48	127.34	75.22	84.49	133.34
CS1. AsssertJ	1.6.1	84.87	133.73	78.46	127.32	75.19	84.46	133.32
CS1. AsssertJ	1.7.0	84.78	133.68	78.42	127.32	75.26	84.45	133.35
CS1. AsssertJ	1.7.1	84.78	133.68	78.42	127.32	75.27	84.45	133.35
CS1. AsssertJ	2.0.0	84.09	133.23	77.96	127.09	75.00	84.01	133.14
CS1. AsssertJ	2.1.0	83.99	133.13	77.72	126.86	74.78	83.86	133.01
CS1. AsssertJ	2.2.0	84.00	133.36	77.72	127.09	74.78	83.89	133.25
CS1. AsssertJ	2.3.0	83.65	132.79	77.47	126.61	74.56	83.63	132.77
CS1. AsssertJ	2.4.0	83.00	132.24	76.84	126.08	73.94	83.09	132.33
CS1. AsssertJ	2.4.1	83.00	132.24	76.84	126.08	73.94	83.09	132.34
CS1. AsssertJ	2.5.0	82.88	132.10	76.71	125.93	73.84	83.05	132.27
CS1. AsssertJ	2.6.0	82.78	132.05	76.55	125.83	73.65	82.94	132.22
CS1. AsssertJ	2.7.0	83.35	132.45	77.23	126.33	74.39	83.65	132.74
CS1. AsssertJ	2.8.0	83.31	132.42	77.19	126.30	74.33	83.61	132.72
CS1. AsssertJ	2.9.0	83.14	132.28	77.25	126.39	74.51	83.76	132.91
CS1. AsssertJ	3.0.0	83.88	132.94	77.66	126.72	74.65	83.74	132.80
CS1. AsssertJ	3.1.0	83.64	132.68	77.35	126.39	74.35	83.55	132.58
CS1. AsssertJ	3.2.0	83.63	132.80	77.33	126.50	74.34	83.55	132.71
CS1. AsssertJ	3.3.0	83.28	132.10	77.05	125.87	74.09	83.26	132.08
CS1. AsssertJ	3.4.0	82.71	131.73	76.50	125.53	73.56	82.80	131.83
CS1. AsssertJ	3.4.1	82.71	131.74	76.51	125.54	73.56	82.80	131.83
CS1. AsssertJ	3.5.0	82.42	131.52	76.21	125.30	73.28	82.60	131.69
CS1. AsssertJ	3.5.1	82.43	131.52	76.21	125.31	73.29	82.60	131.70
CS1. AsssertJ	3.5.2	82.43	131.52	76.21	125.31	73.29	82.60	131.70
CS1. AsssertJ	3.6.0	82.32	131.47	76.06	125.22	73.16	82.50	131.66
CS1. AsssertJ	3.6.1	82.32	131.47	76.06	125.21	73.15	82.50	131.65
CS1. AsssertJ	3.6.2	82.33	131.48	76.07	125.23	73.16	82.51	131.67
CS1. AsssertJ	3.7.0	82.83	131.81	76.67	125.64	73.85	83.15	132.12
CS1. AsssertJ	3.8.0	82.87	131.81	76.70	125.64	73.87	83.18	132.12
CS1. AsssertJ	3.9.0	82.63	131.53	76.75	125.65	74.06	83.32	132.22
CS1. AsssertJ	3.9.1	81.66	130.73	75.92	125.00	73.25	82.60	131.67
CS1. AsssertJ	3.10.0	81.42	130.59	75.68	124.84	73.06	82.40	131.57
CS1. AsssertJ	3.11.0	81.24	130.48	75.38	124.62	72.89	82.59	131.83
CS1. AsssertJ	3.11.1	81.24	130.47	75.38	124.62	72.89	82.59	131.82
CS1. AsssertJ	3.12.0	81.38	130.55	75.52	124.69	73.07	82.66	131.83
CS1. AsssertJ	3.12.1	81.38	130.55	75.52	124.69	73.07	82.66	131.83
CS1. AsssertJ	3.12.2	81.40	130.57	75.53	124.70	73.08	82.67	131.84
CS1. AsssertJ	3.13.0	81.16	130.37	75.23	124.44	72.75	82.34	131.55
CS1. AsssertJ	3.13.1	81.16	130.37	75.23	124.44	72.75	82.34	131.55
CS1. AsssertJ	3.13.2	81.16	130.37	75.23	124.44	72.74	82.34	131.55
CS1. AsssertJ	3.14.0	81.35	130.45	75.43	124.53	72.95	82.53	131.63
CS1. AsssertJ	3.15.0	81.87	130.55	75.88	124.56	73.62	82.82	131.50
CS1. AsssertJ	3.16.0	81.93	130.71	75.94	124.73	73.65	82.92	131.70
CS1. AsssertJ	3.16.1	81.94	130.73	75.95	124.74	73.66	82.93	131.72
CS1. AsssertJ	3.17.0	82.26	130.80	76.24	124.78	73.87	83.23	131.76
CS1. AsssertJ	3.17.1	82.26	130.80	76.24	124.78	73.87	83.23	131.77
CS1. AsssertJ	3.17.2	82.27	130.82	76.25	124.80	73.88	83.24	131.78
CS1. AsssertJ	3.18.0	82.25	130.82	76.22	124.80	73.84	83.22	131.79
CS1. AsssertJ	3.18.1	82.15	130.72	76.18	124.75	73.80	83.18	131.75
CS1. AsssertJ	3.19.0	82.13	130.57	76.17	124.61	73.80	83.15	131.59
CS1. AsssertJ	3.20.0	82.18	130.66	76.16	124.63	73.74	83.15	131.63
CS1. AsssertJ	3.20.1	82.18	130.65	76.15	124.63	73.73	83.15	131.63
CS1. AsssertJ	3.20.2	82.17	130.65	76.14	124.62	73.73	83.14	131.62
CS1. AsssertJ	3.21.0	82.04	130.46	76.05	124.48	73.70	83.06	131.49
CS1. AsssertJ	3.22.0	81.85	130.36	75.85	124.36	73.50	82.88	131.39
CS1. AsssertJ	3.23.0	81.98	130.48	75.98	124.48	73.60	82.99	131.49
CS1. AsssertJ	3.23.1	81.98	130.48	75.98	124.48	73.60	82.99	131.49

Table A3. Index variant values for *CS2.* Caffeine (RQ2.2).

Case Study	Release	$MI_{orig.}$	$MI_{orig.(CM)}$	$MI_{impr.}$	$MI_{impr.(CM)}$	MI_{LOC}	MI_{JH}	$MI_{JH(CM)}$
CS2. Caffeine	1.0	69.38	115.10	62.99	108.71	61.63	68.65	114.37
CS2. Caffeine	1.0.1	70.11	116.11	63.88	109.88	62.48	69.47	115.46
CS2. Caffeine	1.1.0	70.39	116.34	64.19	110.14	62.80	69.76	115.71
CS2. Caffeine	1.2.0	70.39	116.22	64.18	110.01	62.79	69.73	115.57
CS2. Caffeine	1.3.0	69.92	115.21	63.44	108.73	62.12	69.40	114.69
CS2. Caffeine	1.3.1	69.99	115.29	63.52	108.82	62.20	69.47	114.78
CS2. Caffeine	1.3.2	70.41	116.21	63.98	109.78	62.58	69.77	115.57
CS2. Caffeine	1.3.3	70.42	116.21	63.99	109.78	62.58	69.77	115.57
CS2. Caffeine	2.0.0	70.26	115.01	63.94	108.69	62.47	69.71	114.47
CS2. Caffeine	2.0.1	70.34	115.12	64.02	108.81	62.55	69.81	114.59
CS2. Caffeine	2.0.2	70.33	115.15	64.01	108.83	62.52	69.80	114.62
CS2. Caffeine	2.0.3	70.30	115.17	63.98	108.85	62.49	69.76	114.63
CS2. Caffeine	2.1.0	69.94	114.68	63.56	108.31	62.02	69.32	114.07
CS2. Caffeine	2.2.0	69.80	114.56	63.43	108.19	61.90	69.18	113.94
CS2. Caffeine	2.2.1	69.73	114.36	63.34	107.97	61.80	69.13	113.75
CS2. Caffeine	2.2.2	69.72	114.33	63.34	107.94	61.79	69.12	113.72
CS2. Caffeine	2.2.3	69.65	114.23	63.27	107.85	61.73	69.05	113.63
CS2. Caffeine	2.2.4	69.69	114.25	63.31	107.88	61.78	69.09	113.66
CS2. Caffeine	2.2.5	69.68	114.25	63.31	107.88	61.77	69.09	113.66
CS2. Caffeine	2.2.6	69.68	114.25	63.31	107.88	61.77	69.09	113.66
CS2. Caffeine	2.2.7	69.72	114.30	63.35	107.93	61.81	69.12	113.70
CS2. Caffeine	2.3.0	69.62	114.12	63.26	107.76	61.72	69.01	113.51
CS2. Caffeine	2.3.1	69.84	114.37	63.50	108.02	61.93	69.21	113.74
CS2. Caffeine	2.3.2	69.83	114.38	63.49	108.04	61.92	69.21	113.76
CS2. Caffeine	2.3.3	69.82	114.41	63.48	108.08	61.93	69.20	113.80
CS2. Caffeine	2.3.4	69.93	114.36	63.60	108.04	62.00	69.32	113.75
CS2. Caffeine	2.3.5	70.13	114.64	63.83	108.35	62.24	69.53	114.05
CS2. Caffeine	2.4.0	70.36	114.82	64.08	108.53	62.52	69.75	114.21
CS2. Caffeine	2.5.0	70.07	114.42	63.78	108.13	62.24	69.48	113.83
CS2. Caffeine	2.5.1	70.10	114.46	63.81	108.17	62.28	69.51	113.87
CS2. Caffeine	2.5.2	70.04	114.36	63.73	108.06	62.20	69.43	113.75
CS2. Caffeine	2.5.3	70.02	114.35	63.72	108.05	62.18	69.39	113.72
CS2. Caffeine	2.5.4	70.03	114.36	63.72	108.06	62.19	69.39	113.73
CS2. Caffeine	2.5.5	70.05	114.38	63.75	108.08	62.22	69.42	113.74
CS2. Caffeine	2.5.6	70.11	114.44	63.82	108.14	62.29	69.48	113.81
CS2. Caffeine	2.6.0	70.00	114.26	63.68	107.94	62.15	69.37	113.63
CS2. Caffeine	2.6.1	70.02	114.15	63.71	107.84	62.17	69.41	113.54
CS2. Caffeine	2.6.2	69.98	114.12	63.65	107.79	62.11	69.38	113.52
CS2. Caffeine	2.7.0	70.08	113.77	63.66	107.35	62.14	69.46	113.16
CS2. Caffeine	2.8.0	69.77	113.21	63.31	106.75	61.78	69.12	112.55
CS2. Caffeine	2.8.1	69.69	113.11	63.23	106.64	61.67	69.04	112.45
CS2. Caffeine	2.8.2	69.71	113.12	63.25	106.66	61.68	69.06	112.46
CS2. Caffeine	2.8.3	69.76	113.22	63.30	106.76	61.74	69.11	112.57
CS2. Caffeine	2.8.4	69.76	113.21	63.29	106.75	61.73	69.11	112.56
CS2. Caffeine	2.8.5	69.78	113.23	63.32	106.78	61.76	69.14	112.59
CS2. Caffeine	2.8.6	69.59	112.94	63.08	106.43	61.49	68.87	112.22
CS2. Caffeine	2.8.7	69.42	112.64	62.90	106.12	61.31	68.68	111.91
CS2. Caffeine	2.8.8	69.42	112.64	62.90	106.12	61.30	68.68	111.91
CS2. Caffeine	2.9.0	69.35	112.62	62.83	106.10	61.24	68.67	111.94
CS2. Caffeine	2.9.1	69.28	112.52	62.76	105.99	61.17	68.59	111.82
CS2. Caffeine	2.9.2	69.18	112.37	62.64	105.82	61.05	68.46	111.65
CS2. Caffeine	2.9.3	69.02	112.11	62.45	105.55	60.87	68.29	111.39
CS2. Caffeine	3.0.0	69.70	112.56	63.28	106.13	61.77	68.94	111.79
CS2. Caffeine	3.0.1	69.69	112.53	63.27	106.11	61.75	68.93	111.77
CS2. Caffeine	3.0.2	69.66	112.91	63.23	106.48	61.71	68.85	112.10
CS2. Caffeine	3.0.3	70.34	114.05	63.97	107.68	62.42	69.43	113.14
CS2. Caffeine	3.0.4	71.32	115.60	65.16	109.44	63.50	70.45	114.74
CS2. Caffeine	3.0.5	71.30	115.51	65.14	109.35	63.48	70.44	114.64

Table A3. *Cont.*

Case Study	Release	$MI_{orig.}$	$MI_{orig.(CM)}$	$MI_{impr.}$	$MI_{impr.(CM)}$	MI_{LOC}	MI_{JH}	$MI_{JH(CM)}$
CS2. Caffeine	3.0.6	69.75	113.75	63.36	107.37	61.80	68.47	112.47
CS2. Caffeine	3.1.0	69.37	113.26	63.06	106.95	61.50	68.22	112.11
CS2. Caffeine	3.1.1	69.49	113.37	63.18	107.06	61.66	68.26	112.15

Table A4. Index variant values for *CS3.* Joda-Time (RQ2.2).

Case Study	Release	$MI_{orig.}$	$MI_{orig.(CM)}$	$MI_{impr.}$	$MI_{impr.(CM)}$	MI_{LOC}	MI_{JH}	$MI_{JH(CM)}$
CS3. Joda-Time	0.9	59.04	104.72	52.37	98.05	50.51	57.43	103.12
CS3. Joda-Time	0.9.5	58.99	101.47	51.45	93.93	50.84	56.77	99.25
CS3. Joda-Time	1.0	54.77	104.65	46.70	96.57	44.74	50.68	100.55
CS3. Joda-Time	1.1	53.45	103.29	45.37	95.22	43.20	49.43	99.28
CS3. Joda-Time	1.2	55.59	105.39	47.81	97.61	45.61	51.77	101.57
CS3. Joda-Time	1.2.1	52.22	102.02	44.12	93.92	41.92	48.04	97.84
CS3. Joda-Time	1.3	53.49	103.30	45.52	95.33	43.15	49.37	99.18
CS3. Joda-Time	1.4	52.98	102.84	44.94	94.80	42.53	48.79	98.65
CS3. Joda-Time	1.5	52.68	102.51	44.64	94.47	42.17	48.53	98.36
CS3. Joda-Time	1.5.1	52.67	102.50	44.63	94.46	42.16	48.52	98.35
CS3. Joda-Time	1.5.2	52.64	102.47	44.61	94.44	42.13	48.50	98.33
CS3. Joda-Time	1.6.0	52.38	102.20	44.34	94.16	41.84	48.23	98.05
CS3. Joda-Time	1.6.1	52.34	102.16	44.30	94.12	41.80	48.19	98.00
CS3. Joda-Time	1.6.2	52.34	102.16	44.30	94.12	41.80	48.18	98.00
CS3. Joda-Time	2.0	47.75	97.60	39.28	89.13	36.60	43.09	92.94
CS3. Joda-Time	2.1	47.67	97.52	39.21	89.05	36.52	43.01	92.85
CS3. Joda-Time	2.2	47.52	97.36	39.04	88.88	36.34	42.84	92.67
CS3. Joda-Time	2.3	47.49	97.31	39.00	88.83	36.31	42.82	92.64
CS3. Joda-Time	2.4	47.60	97.41	39.13	88.94	36.44	42.93	92.74
CS3. Joda-Time	2.5	47.51	97.31	39.04	88.83	36.37	42.84	92.64
CS3. Joda-Time	2.6	47.53	97.32	39.05	88.84	36.38	42.85	92.65
CS3. Joda-Time	2.7	47.53	97.33	39.06	88.85	36.39	42.86	92.65
CS3. Joda-Time	2.8	47.53	97.32	39.06	88.85	36.39	42.86	92.65
CS3. Joda-Time	2.8.1	47.53	97.32	39.06	88.85	36.39	42.86	92.65
CS3. Joda-Time	2.8.2	47.53	97.32	39.06	88.85	36.39	42.86	92.65
CS3. Joda-Time	2.9	47.43	97.21	38.95	88.73	36.28	42.75	92.53
CS3. Joda-Time	2.9.1	47.42	97.21	38.94	88.73	36.28	42.74	92.52
CS3. Joda-Time	2.9.2	47.40	97.19	38.93	88.71	36.25	42.72	92.50
CS3. Joda-Time	2.9.3	47.40	97.18	38.92	88.70	36.25	42.72	92.50
CS3. Joda-Time	2.9.4	47.48	97.26	39.01	88.79	36.34	42.81	92.59
CS3. Joda-Time	2.9.5	47.42	97.19	38.94	88.71	36.27	42.74	92.51
CS3. Joda-Time	2.9.6	47.41	97.18	38.93	88.70	36.26	42.73	92.50
CS3. Joda-Time	2.9.7	47.41	97.18	38.93	88.70	36.26	42.73	92.51
CS3. Joda-Time	2.9.8	47.43	97.21	38.96	88.73	36.29	42.76	92.53
CS3. Joda-Time	2.9.9	47.43	97.21	38.96	88.73	36.29	42.76	92.53
CS3. Joda-Time	2.10	47.41	97.18	38.93	88.70	36.27	42.74	92.51
CS3. Joda-Time	2.10.1	47.40	97.17	38.92	88.69	36.25	42.72	92.49
CS3. Joda-Time	2.10.2	47.39	97.16	38.91	88.68	36.24	42.72	92.48
CS3. Joda-Time	2.10.3	47.36	97.12	38.89	88.65	36.22	42.69	92.45
CS3. Joda-Time	2.10.4	47.36	97.12	38.89	88.65	36.22	42.69	92.45
CS3. Joda-Time	2.10.5	47.36	97.12	38.89	88.65	36.22	42.69	92.45
CS3. Joda-Time	2.10.6	47.36	97.12	38.89	88.65	36.22	42.69	92.45
CS3. Joda-Time	2.10.7	47.28	97.03	38.80	88.55	36.13	42.59	92.34
CS3. Joda-Time	2.10.8	47.28	97.03	38.80	88.55	36.13	42.59	92.34
CS3. Joda-Time	2.10.9	47.28	97.03	38.80	88.55	36.13	42.59	92.34
CS3. Joda-Time	2.10.10	47.28	97.03	38.80	88.55	36.13	42.59	92.34
CS3. Joda-Time	2.10.11	47.27	97.02	38.79	88.54	36.12	42.59	92.34
CS3. Joda-Time	2.10.12	47.27	97.02	38.79	88.54	36.12	42.59	92.34
CS3. Joda-Time	2.10.13	47.27	97.02	38.79	88.54	36.12	42.59	92.34

Table A4. *Cont.*

Case Study	Release	$MI_{orig.}$	$MI_{orig.(CM)}$	$MI_{impr.}$	$MI_{impr.(CM)}$	MI_{LOC}	MI_{JH}	$MI_{JH(CM)}$
CS3. Joda-Time	2.10.14	47.27	97.02	38.79	88.54	36.12	42.59	92.34
CS3. Joda-Time	2.11.0	46.93	96.61	38.44	88.11	35.55	42.54	92.21
CS3. Joda-Time	2.11.1	46.93	96.61	38.44	88.11	35.55	42.54	92.21

References

1. *ISO/IEC/IEEE 14764:2022*; Software engineering—Software Life Cycle Processes—Maintenance. International Organization for Standardization: Geneva, Switzerland, 2022.
2. Christa, S.; Madhusudhan, V.; Suma, V.; Rao, J.J. Software Maintenance: From the Perspective of Effort and Cost Requirement. In Proceedings of the International Conference on Data Engineering and Communication Technology, Maharashtra, India, 10–11 March 2017; Satapathy, S., Bhateja, V., Joshi, A., Eds.; Springer: Singapore, 2017; pp. 759–768.
3. Granja-Alvarez, J.C.; Barranco-García, M.J. A Method for Estimating Maintenance Cost in a Software Project: A Case Study. *J. Softw. Maint.* **1997**, *9*, 161–175. [CrossRef]
4. Ren, Y.; Xing, T.; Chen, X.; Chai, X. Research on Software Maintenance Cost of Influence Factor Analysis and Estimation Method. In Proceedings of the 2011 3rd International Workshop on Intelligent Systems and Applications, Wuhan, China, 28–29 May 2011; pp. 1–4. [CrossRef]
5. *ISO/IEC 25010:2011*; Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (SQuaRE)—System and Software Quality Models. International Organization for Standardization: Geneva, Switzerland, 2017.
6. Alsolai, H.; Roper, M. A systematic literature review of machine learning techniques for software maintainability prediction. *Inf. Softw. Technol.* **2020**, *119*, 106214. [CrossRef]
7. Riaz, M.; Mendes, E.; Tempero, E. A systematic review of software maintainability prediction and metrics. In Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, Lake Buena Vista, FL, USA, 15–16 October 2009; pp. 367–377. [CrossRef]
8. Oman, P.; Hagemeister, J. Construction and testing of polynomials predicting software maintainability. Oregon Workshop on Software Metrics. *J. Syst. Softw.* **1994**, *24*, 251–266. [CrossRef]
9. Coleman, D.; Ash, D.; Lowther, B.; Oman, P. Using metrics to evaluate software system maintainability. *Computer* **1994**, *27*, 44–49. [CrossRef]
10. Welker, K.D.; Oman, P.W.; Atkinson, G.G. Development and Application of an Automated Source Code Maintainability Index. *J. Softw. Maint.* **1997**, *9*, 127–159. [CrossRef]
11. Najm, N. Measuring Maintainability Index of a Software Depending on Line of Code Only. *IOSR J. Comput. Eng.* **2014**, *16*, 64–69. [CrossRef]
12. Microsoft. Visual Studio. 2021. Available online: https://visualstudio.microsoft.com/ (accessed on 20 December 2021).
13. Virtual Machinery. MI and MINC—Maintainability Index. 2021. Available online: http://www.virtualmachinery.com/sidebar4.htm (accessed on 20 December 2021).
14. Kaur, K.; Singh, H. Determination of Maintainability Index for Object Oriented Systems. *Determ. Maintainab. Index Object Oriented Syst.* **2011**, *36*, 1–6. [CrossRef]
15. Kaur, A.; Kaur, K.; Pathak, K. A proposed new model for maintainability index of open source software. In Proceedings of the 3rd International Conference on Reliability, Infocom Technologies and Optimization, Noida, India, 8–10 October 2014; pp. 1–6. [CrossRef]
16. Misra, S.C. Modeling Design/Coding Factors That Drive Maintainability of Software Systems. *Softw. Qual. J.* **2005**, *13*, 297–320.
17. Madhwaraj, K.G. Empirical comparison of two metrics suites for maintainability prediction in packages of object-oriented systems: A case study of open source software. *J. Comput. Sci.* **2014**, *10*, 2330–2338. [CrossRef]
18. Welker, K.D. The software maintainability index revisited. *CrossTalk* **2001**, *14*, 18–21.
19. Sjøberg, D.I.K.; Anda, B.; Mockus, A. Questioning software maintenance metrics: A comparative case study. In Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, Lund, Sweden, 19–20 September 2012; pp. 107–110. [CrossRef]
20. Counsell, S.; Liu, X.; Eldh, S.; Tonelli, R.; Marchesi, M.; Concas, G.; Murgia, A. Re-visiting the 'Maintainability Index' Metric from an Object-Oriented Perspective. In Proceedings of the 2015 41st Euromicro Conference on Software Engineering and Advanced Applications, Madeira, Portugal, 26–28 August 2015; pp. 84–87. [CrossRef]
21. Seref, B.; Tanriover, O. Software code maintainability: A literature review. *Int. J. Softw. Eng. Appl.* **2016**, *7*, 3. [CrossRef]
22. Ganpati, A.; Kalia, A.; Singh, H. A comparative study of maintainability index of open source software. *Int. J. Emerg. Technol. Adv. Eng.* **2012**, *2*, 228–230.
23. Fedoseev, K.; Askarbekuly, N.; Uzbekova, E.; Mazzara, M. Application of Data-Oriented Design in Game Development. *J. Phys. Conf. Ser.* **2020**, *1694*, 106218. [CrossRef]
24. Molnar, A.; Motogna, S. Discovering Maintainability Changes in Large Software Systems. In Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement, Gothenburg, Sweden, 25–27 October 2017; Association for Computing Machinery: New York, NY, USA, 2017; pp. 88–93. [CrossRef]

25. Papamichail, M.D.; Symeonidis, A.L. A generic methodology for early identification of non-maintainable source code components through analysis of software releases. *Inf. Softw. Technol.* **2020**, *118*, 106218. [CrossRef]
26. Kencana, G.H.; Saleh, A.; Darwito, H.A.; Rachmadi, R.R.; Sari, E.M. Comparison of Maintainability Index Measurement from Microsoft CodeLens and Line of Code. In Proceedings of the 2020 7th International Conference on Electrical Engineering, Computer Sciences and Informatics (EECSI), Yogyakarta, Indonesia, 1–2 October 2020; pp. 235–239.
27. Şanlıalp, İ.; Öztürk, M.M.; Yiğit, T. Energy Efficiency Analysis of Code Refactoring Techniques for Green and Sustainable Software in Portable Devices. *Electronics* **2022**, *11*, 442. [CrossRef]
28. Zhou, Y.; Xu, B. Predicting the maintainability of open source software using design metrics. *Wuhan Univ. J. Nat. Sci.* **2008**, *13*, 14–20. [CrossRef]
29. Chowdhury, S.; Holmes, R.; Zaidman, A.; Kazman, R. Revisiting the debate: Are code metrics useful for measuring maintenance effort? *Empir. Softw. Eng.* **2022**, *27*, 1–31. [CrossRef]
30. Strečanský, P.; Chren, S.; Rossi, B. 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, 30 March–3 April 2020; Association for Computing Machinery: New York, NY, USA, 2020; pp. 121–124. [CrossRef]
31. Arisholm, E.; Briand, L.C.; Johannessen, E.B. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *J. Syst. Softw.* **2010**, *83*, 2–17. [CrossRef]
32. Mauša, G.; Galinac Grbac, T. Co-evolutionary multi-population genetic programming for classification in software defect prediction: An empirical case study. *Appl. Soft Comput.* **2017**, *55*, 331–351. [CrossRef]
33. Gradišnik, M.; Beranič, T.; Karakatič, S. Impact of historical software metric changes in predicting future maintainability trends in open-source software development. *Appl. Sci.* **2020**, *10*, 4624. [CrossRef]
34. Kaur, L.; Mishra, A. A comparative analysis of evolutionary algorithms for the prediction of software change. In Proceedings of the 2018 International Conference on Innovations in Information Technology (IIT), Al Ain, United Arab Emirates, 18–19 November 2018; pp. 187–192. [CrossRef]
35. Reddy, B.R.; Ojha, A. Performance of maintainability index prediction models: A feature selection based study. *Evol. Syst.* **2019**, *10*, 179–204. [CrossRef]
36. Bray, M.; Brune, K.; Fisher, D.A.; Foreman, J.; Gerken, M. *C4 Software Technology Reference Guide-A Prototype*; Technical Report; Software Engineering Institute—Carnegie Mellon University: Pittsburgh, PA, USA, 1997.
37. GitHub. The 2021 State of the Octoverse. 2021. Available online: https://octoverse.github.com/ (accessed on 20 December 2021).
38. Stack Overflow. Stack Overflow Developer Survey. 2021. Available online: https://insights.stackoverflow.com/survey/2021 (accessed on 20 December 2021).
39. Puntigam, F. Interfaces of Active Objects with Internal Concurrency. In Proceedings of the 1st International Workshop on Distributed Objects for the 21st Century, Genova, Italy, 6–10 July 2009; Association for Computing Machinery: New York, NY, USA, 2009. [CrossRef]
40. Murthy, A.R.C.; Palani, G.; Iyer, N.R. Object-oriented programming paradigm for damage tolerant evaluation of engineering structural components. *Adv. Eng. Softw.* **2011**, *42*, 12–24. [CrossRef]
41. Brito e Abreu, F.; Melo, W. Evaluating the impact of object-oriented design on software quality. In Proceedings of the 3rd International Software Metrics Symposium, Berlin, Germany, 25–26 March 1996; pp. 90–99. [CrossRef]
42. Coleman, D. Assessing maintainability. In Proceedings of the Software Engineering Productivity Conference, Salt Lake City, UT, USA, 7 April 1992; pp. 525–532.
43. Elmidaoui, S.; Cheikhi, L.; Idri, A.; Abran, A. Empirical studies on software product maintainability prediction: A systematic mapping and review. *E-Inform. Softw. Eng. J.* **2019**, *13*, 141–202.
44. Zighed, N.; Bounour, N.; Seriai, A.D. Comparative Analysis of Object-Oriented Software Maintainability Prediction Models. *Found. Comput. Decis. Sci.* **2018**, *43*, 359–374. [CrossRef]
45. Ardito, L.; Coppola, R.; Barbato, L.; Verga, D. A tool-based perspective on software code maintainability metrics: A systematic literature review. *Sci. Program.* **2020**, *2020*, 8840389. [CrossRef]
46. Virtual Machinery. JHawk. 2021. Available online: http://www.virtualmachinery.com/jhawkprod.htm (accessed on 20 December 2021).
47. Lacchia, M. Radon. 2021. Available online: https://radon.readthedocs.io/ (accessed on 20 December 2021).
48. Verifysoft Technology. Testwell CMT++/CMTJava. 2021. Available online: https://www.verifysoft.com/en_cmtx.html (accessed on 20 December 2021).
49. Li, W.; Henry, S. Object-oriented metrics that predict maintainability. Object-Oriented Software. *J. Syst. Softw.* **1993**, *23*, 111–122. [CrossRef]
50. Malhotra, R.; Khanna, M. Particle swarm optimization-based ensemble learning for software change prediction. *Inf. Softw. Technol.* **2018**, *102*, 65–84. [CrossRef]
51. Elish, M.O.; Aljamaan, H.; Ahmad, I. Three empirical studies on predicting software maintainability using ensemble methods. *Soft Comput.* **2015**, *19*, 2511–2524. [CrossRef]
52. Bandi, R.; Vaishnavi, V.; Turk, D. Predicting maintenance performance using object-oriented design complexity metrics. *IEEE Trans. Softw. Eng.* **2003**, *29*, 77–87. [CrossRef]
53. Fioravanti, F.; Nesi, P. Estimation and prediction metrics for adaptive maintenance effort of object-oriented systems. *IEEE Trans. Softw. Eng.* **2001**, *27*, 1062–1084. [CrossRef]

54. Hayes, J.; Patel, S.; Zhao, L. A metrics-based software maintenance effort model. In Proceedings of the 8th European Conference on Software Maintenance and Reengineering, Tampere, Finland, 24–26 March 2004; pp. 254–258. [CrossRef]
55. De Lucia, A.; Pompella, E.; Stefanucci, S. Assessing effort estimation models for corrective maintenance through empirical studies. *Inf. Softw. Technol.* **2005**, *47*, 3–15. [CrossRef]
56. Dahiya, S.S.; Chhabra, J.K.; Kumar, S. Use of genetic algorithm for software maintainability metrics' conditioning. In Proceedings of the 15th International Conference on Advanced Computing and Communications (ADCOM 2007), Guwahati, India, 18–21 December 2007; pp. 87–92. [CrossRef]
57. Sharma, A.; Grover, P.; Kumar, R. Predicting maintainability of component-based systems by using fuzzy logic. In Proceedings of the International Conference on Contemporary Computing, Noida, India, 17–19 August 2009; Springer: Berlin/Heidelberg, Germany, 2009; pp. 581–591.
58. Dubey, S.K.; Rana, A. A fuzzy approach for evaluation of maintainability of object oriented software system. *Int. J. Comput. Appl.* **2012**, *49*, 1–6.
59. Pratap, A.; Chaudhary, R.; Yadav, K. Estimation of software maintainability using fuzzy logic technique. In Proceedings of the 2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT), Ghaziabad, India, 7–8 February 2014; pp. 486–492. [CrossRef]
60. Revilla, M.A. Correlations between Internal Software Metrics and Software Dependability in a Large Population of Small C/C++ Programs. In Proceedings of the 18th IEEE International Symposium on Software Reliability (ISSRE '07), Trollhattan, Sweden, 5–9 November 2007; pp. 203–208. [CrossRef]
61. Microsoft. Visual Studio—Maintainability Index. 2021. Available online: https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-maintainability-index-range-and-meaning (accessed on 20 December 2021).
62. Molnar, A.J.; Motogna, S. A Study of Maintainability in Evolving Open-Source Software. In Proceedings of the Evaluation of Novel Approaches to Software Engineering: 15th International Conference, ENASE 2020, Prague, Czech Republic, 5–6 May 2020; Revised Selected Papers 15; Springer: Berlin/Heidelberg, Germany, 2021; pp. 261–282.
63. Chowdhury, S.A.; Uddin, G.; Holmes, R. An Empirical Study on Maintainable Method Size in Java. In Proceedings of the 19th International Conference on Mining Software Repositories, Pittsburgh, PA, USA, 23–24 May 2022; Association for Computing Machinery: New York, NY, USA, 2022; pp. 252–264. [CrossRef]
64. Prabowo, G.; Suryotrisongko, H.; Tjahyanto, A. A Tale of Two Development Approach: Empirical Study on The Maintainability and Modularity of Android Mobile Application with Anti-Pattern and Model-View-Presenter Design Pattern. In Proceedings of the 2018 International Conference on Electrical Engineering and Informatics (ICELTICs), Banda Aceh, Indonesia, 19–20 September 2018; pp. 149–154. [CrossRef]
65. Singh, N.; Singh, D.P.; Pant, B.; Tiwari, U.K. μBIGMSA-Microservice-Based Model for Big Data Knowledge Discovery: Thinking Beyond the Monoliths. *Wirel. Pers. Commun.* **2021**, *116*, 2819–2833. [CrossRef]
66. Wilson, A.; Wedyan, F.; Omari, S. An Empirical Evaluation and Comparison of the Impact of MVVM and MVC GUI Driven Application Architectures on Maintainability and Testability. In Proceedings of the 2022 International Conference on Intelligent Data Science Technologies and Applications (IDSTA), San Antonio, TX, USA, 5–7 September 2022; pp. 101–108. [CrossRef]
67. De Stefano, M.; Iannone, E.; Pecorelli, F.; Tamburri, D.A. Impacts of software community patterns on process and product: An empirical study. *Sci. Comput. Program.* **2022**, *214*, 102731. [CrossRef]
68. Lavazza, L.; Abualkishik, A.Z.; Liu, G.; Morasca, S. An empirical evaluation of the "Cognitive Complexity" measure as a predictor of code understandability. *J. Syst. Softw.* **2023**, *197*, 111561. [CrossRef]
69. Cohen, J. *Statistical Power Analysis for the Behavioral Sciences*; Routledge Academic: Milton Park, UK, 1988.
70. Sawilowsky, S.S. New effect size rules of thumb. *J. Mod. Appl. Stat. Methods* **2009**, *8*, 26. [CrossRef]
71. Kerby, D.S. The simple difference formula: An approach to teaching nonparametric correlation. *Compr. Psychol.* **2014**, *3*, 11-IT. [CrossRef]
72. Schober, P.; Boer, C.; Schwarte, L.A. Correlation coefficients: Appropriate use and interpretation. *Anesth. Analg.* **2018**, *126*, 1763–1768. [CrossRef]
73. Hout, M.C.; Papesh, M.H.; Goldinger, S.D. Multidimensional scaling. *Wiley Interdiscip. Rev. Cogn. Sci.* **2013**, *4*, 93–103. [CrossRef]
74. Vijaya; Sharma, S.; Batra, N. Comparative Study of Single Linkage, Complete Linkage, and Ward Method of Agglomerative Clustering. In Proceedings of the 2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon), Faridabad, India, 14–16 February 2019; pp. 568–573. [CrossRef]
75. Heričko, T.; Šumak, B. Analyzing Linter Usage and Warnings Through Mining Software Repositories: A Longitudinal Case Study of JavaScript Packages. In Proceedings of the 2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO), Opatija, Croatia, 23–27 May 2022; pp. 1375–1380. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.