Science, Engineering and Technology Vol. 3, No. 2, pp. 97-105
www.setjournal.com *https://doi.org/10.54327/set2023/v3.i2.73*

A Source-Code Maintainability Evaluation Model for Software Products

Hayatou Oumarou, Kolyang

Department of Computer Science, Higher Teachers' Training College, The University of Maroua, PO Box 55 Maroua, Cameroon

Abstract

The maintainability index (MI) has been proposed to calculate a single number which expresses the maintainability of a system. This article presents a model for evaluating the maintainability of software products. The model improves the shortcomings observed in the maintainability assessment approaches in the quality assessment models SQuaRE (ISO25000), ISO 9126, Squale and the FCM standard. Its main innovation is to take into account the importance of entities in the system when calculating the maintainability score. This implies that the same type of defect will have a different score depending on the entity presenting it. Seven experts with several years of experience evaluated the model. They confirmed the effectiveness and usability of the model. Then, we compared our model with the Squale maintainability index and the classical maintainability index. The results show no correlation between these models. The implications are that each method gives a slightly different view of maintainability.

Keywords: *maintainability, software quality, metric, evaluation model.*

1. Introduction

Nowadays, organizations are more and more concerned about the quality of the software they use. The numerous failures and errors, the evolution of the software to meet future expectations, and the multitude of existing hardware and software platforms make them conscious of being concerned about the software quality. Therefore, it is essential to assess the quality of their software. For example, on June 15, 2022, a software failure obliged the closure of Swiss airspace from 04:00 until 08:30. Thousands of passengers were affected, and hundreds of flights were canceled or diverted to neighboring airports causing huge losses [1]. In such cases, urgent maintenance is necessary. Studies have shown that 50-90% of software effort goes into maintaining systems. Maintainability strongly influences maintenance teams' productivity [2]. The reasons for this very high cost are multiple and often misunderstood. Most of the software development time is devoted to the maintenance phase. Improving an application's maintainability quality positively influences the latter's cost [2]. Thus, one of the main objectives of software engineers is the development of easily maintainable software. The maintainability of software influences its overall quality. With maintainable software, it is easy to modify parts, solve stakeholder problems very quickly and manage the software efficiently [3].

To control the software quality, measuring and evaluating its quality characteristics is essential. The maintenance assessment is usually part of the quality models that assess software quality. This evaluation uses measurements, generally mathematical models, considering the expected aspects. The maintainability is an essential part to software quality. A quality model relates the external quality of software to its internal quality. Quality models combine the values of metrics and raw data in a well-defined way to ease quality analysis. Researchers have proposed measures, such as maintainability index and quality models. For example, McCall's FCM (factor, criterion and metric) model [4], the Boehm model, the Dromey model, the ISO 9126 model [5], the SQuaRE model [6], the Squale model [7], the Squash model [8]. Most of these models offer guidelines for evaluating the quality of a software product but often ignore certain essential aspects to measure. Assessing the

Corresponding author: *Hayatou Oumarou (oumarou.hayatou@univ-maroua.cm)*
Received: *4 March 2023*; Revised: *28 May 2023*; Accepted: *2 June 2023*; Published: *30 June 2023*

quality of software from metrics is not always easy. A metric is defined and used for specific software components (i.e., the SLOC (Source Line of Code) metric for a method or the DIT (Depth of Inheritance Tree) metric for class). Principles or rules to be measured usually call for using several metrics belonging to different ranges. Determining a score can involve solving two problems:

- *Compose* metrics that are different from each other.
- *Aggregate* the results to find an overall score for a criterion while retaining the information provided by each raw score.

We cannot derive some measures from metrics, such as assessing documentation quality. Instead, human expertise is used to evaluate such features.

Based on these observations, we propose a new model for measuring maintainability. To achieve this general objective, we formulate four specific research objectives: (1) Identify measurable maintainability characteristics of software; (2) find the appropriate metrics used to measure each characteristic; (3) define the mathematical formulas or algorithms to be used for the calculation of the scores; (4) validate and evaluate the formalized model by applying it to concrete case studies.

The rest of the paper falls as follows: Section 2: State of the art. This section presents related works. Section 3: Proposed approach. This section describes the proposed maintainability model. It highlights the architecture, and the formulas used to calculate the scores in our model. Section 4: Empirical assessment. In this section, we validate the proposed model through case studies. Section 5: Conclusion.

2. State-of-the-art

Researchers propose many software maintainability evaluation models. This paper focuses on the Maintainability Index (MI) and Squale Maintainability Index (SQI). There are several reasons for choosing these models. On the one hand, the Maintainability Index is a popular model proposed. Then the Squale model implements the ISO9126 standard into practice. Our model has its origins in the limits of these cited models. These approaches implemented by tools are based on metrics, so they assume a similar effort in collecting data.

The idea of the Maintainability Index (MI) was first presented in 1992 by Oman and Heidegger [9]. They present a study on 60 measurements and aggregate some of them to evaluate the maintainability of software. They then proposed the maintainability index evaluated by the following formula:

$$MI=171-5.2\ln(HV)-0.23(CC)-16.2\ln(LoC)+0.99(CMT)$$

where

- *HV* is the average Halstead volume per module,
- *CC* is an average cyclomatic complexity per module,
- *LoC* is the average lines of code per module and
- *CMT* is an average of comment lines per module.

Software specialist consider the maintainability index as controversial measure, despite its popularity. They criticized the unclear explanation of formula, the use of simple averages per file, the same formula till 1994, and ambiguous connection between the results and specific metrics in the code source. Several formulae are derived from MI and are still used in some popular code editors (e.g., Microsoft Visual Studio) [9].

The Squale quality model [10], designed by Qualixo and Air-France KLM, comprises four layers, divided into two levels: (i) the conceptual level defining the main assessed principles of quality. It comprises factors and criteria. The technical level outlines basic rules and associated measures. Based on ISO 9126 and SQuaRE standards, the model incorporates seven factors according to empirical validations performed during the Squale and Squash research projects. For this model, the maintainability (which measures the ability of software to facilitate the location and correction of residual errors), is evaluated by the Squale Maintainability Index (SMI). This, deals with the correction of non-conformity defects about the specifications and not of defects resulting from a poor expression of needs. The following criteria: simplicity, interdependence, homogeneity and technical documentation. Authors use formulas to combine and aggregate source code metrics. They also use fuzzy logic formulas, allowing quality evaluation from measurements, not from metrics (raw data). The rating value is in the interval [0, 3]. Squale is well-structured, it clearly defines the quality factors. These factors are decomposed into simple and easily measurable properties. This model links the quality factors, and the metrics used for their evaluations. However, considering

maintainability as the ability to locate and correct errors without considering the criticality of the localization limits its evaluation. Similarly, the model does not consider the capacity for modification and evolution.

SQuaRE model derived from Squale, is a family of standards which divides software quality characteristics into two quality models. Quality of use measures how well a product or system meets user requirements to achieve a specific goal; and product quality, measures product characteristics related to software properties. SQuaRE model has six main sections. Quality measurement is discussed in section three and section five looks at quality assessment. The SQuaRE standard is gradually replacing existing standards related to software quality, such as the ISO 9126 quality model. Al-Kilidar [11] and [12] revealed limitations of the ISO 9126 and SQuaRE (ISO25000) models for the quality/maintainability assessment of software products. Critics of this standard come from an experiment involving design in pairs. One severe criticism they give is that the 9126 standard does not provide guidance, heuristics, empirical rules, or any other way indicate how to use metrics, weight metrics, or even put them together simply. However, by analyzing this model, we realize that it also treats the defects of the same nature uniformly.

Our main goal is to evaluate maintainability through a source code quality model. The proposed model combines aspects of the ISO 9126, SQuaRE, Squale and Squash models (see Table 1). We identified the model sub-factors and criteria based on the limitations of the mentioned models above. It aims to overcome certain limitations of these models. Separating the characteristics of maintainability and capacity for evolution, and not considering the criticality of fault location (a fault in a method used by several others differs from a fault in a less used method). Evolutionary capacity must be integrated with evolutive maintainability for a complete maintainability evaluation [13].

By considering the modifiability sub-characteristic of the SQuaRE model as being equal to the sub-characteristics fusion: changeability and stability of the ISO 9126 model [14], and considering that, the analyzability sub-characteristic of the ISO 9126 model, can be evaluated using the criteria: simplicity, technical documentation and interdependence of the Squash model.

Table 1. Summary of the criteria for the "Maintainability"

	Models			
Criteria/sub-characteristics	ISO 9126	SQuaRE (ISO 25000)	Squale	Squash
Analyzability	×			
Changeability	×			
technical documentation			×	×
Homogeneity			×	×
interdependence			×	×
Editability		×		
Modularity	×	×		
Reusability	×	×		×
Simplicity			×	×
Stability	×			
Testability	×	×		

3. Proposed Approach

To evaluate the maintainability, we break it down into criteria, practices, and measures. Criteria defines the high level of abstraction of maintainability fields. These fields are changeability, technical documentation, interdependence, homogeneity, modelization, modularity, reusability, simplicity, stability, and testability. Each field is evaluated through practices or measures. Practices and measures are concrete and constitute the technical level of our model. We have five practices: Code practices, norms and standards practices, testing practices, documentation practices, and model practices. Measures are raw data collected from metrics or documents attached to the project. We break down these measures into five major groups: Code metrics (i.e., the number of lines of code), Test metrics (i.e., coverage rate), Model metrics (i.e., number of attributes, number of public fields), Coding conventions and Raw data. Raw data corresponds to the audits/assessments performed by experts on the application documentation. We assign each measurement to the entity it measures (class, method, variable, etc.) and aggregate them at the level of the project. Our approach can be summed up in four major steps:

1) Calculation of the importance of an entity;
2) Calculation of an entity's score for a practice;
3) Aggregation of the entity's scores for a practice in order to get the score of the practice at the project level;
4) Aggregation of the project practices weighted scores in order to get the maintainability score of the project.

3.1. Calculation of the importance of an entity

To assess the maintainability of an entity, we use its importance. The importance of an entity is given by interactions it has with its environment. For this first version, we consider variables, methods and classes that we call candidate. The algorithm uses a mechanism similar to Google's PageRank. For the sake of readability, we present the algorithm in three parts. The main algorithm is Importance. It takes the list of candidates and returns a list of pairs (candidate, score) where score is the maintainability score of the candidate. It uses the Neighborhood and Score algorithms which calculate the importance and the score of the candidate.

Algorithm Importance: return a list of couples (candidate, score)

INPUT: *L*, the list of candidates
LOCAL: *chg*, the list of couple (l, c) where l in L and c the computed score
    For each l **in** L
        Chg add (l, score (l))
    End for
    For *i=1* **to** *threshold* **do**
      For each (l, c) **in** chg
        (l,c)<-(l, (c + **SUM** $_{l' \text{ In } Neighbor\ (l)}$ $score(l')$) /1+
                    (c + **SUM** $_{l' \text{ In } Neighbor\ (l)}$ $score(l')$))
      End For
    End for
  return *Chg*

Algorithm Neighbor: find neighbor of a candidate

INPUT: *l*, a Candidate
RETURN: *N*, a list of neighbor of l
    Case type of l:
      Method : N <- senders(l) +methodSends()
      Class : N <- AllRefInside (l) + AllRefOutside (l)
    End Case
  return *N*

Algorithm score: give a score to a candidate

INPUT: *l*, a Candidate
RETURN: *c*, the score for l
    c <- # neighbor(l) / 1+ #neighbor (l)
  return *c*

3.2. Calculation of an entity's score for a practice and aggregation

The practice evaluation in our model follows the Squash scoring model requirements. They are derived from Squash procedures and formulas to compute the different practices scores (code practices, norms, and standards practices, test practices, model practices and documentary practices).

We calculate scores in the interval [0; 3] like in Squash according to the following meanings:

- between 0 and 1, the objective is not reached;
- between 1 and 2, the minimum was reached, but with issues;
- between 2 and 3, the objective is reached.

Formula are specific to each practices.

3.2.1. Code practices scores

Code practices rely on metrics from code metrics. First, we transform each value computed metric in the interval [0; 3]. Each practice, therefore, gets a score per component, called an individual score. Coding practices are intimately linked to code smells [15]. They are: Self-description, Spaghetti codes, Class cohesion, Copy paste, Swiss knife, Cyclic dependency, Level of abstraction and stability, Number of methods, Inheritance depth, Class specialization, Size of a method, Comment rate.

For the criterion number of methods, for example, a study on 2080 Java applications taken randomly from SourceForge [16] found 938,779 methods defined in 270,973 Java classes. On average, there are 3.5 methods per class (median equal to four) with a range of 1 to 1175 methods per class. We apply the following scale for this criterion:

$$note = \begin{cases} 3 \; if \; n < 5 \\ 3 - (n - 5) * 0{,}25 \end{cases} \quad (1)$$

Since the entities / candidates of the system do not have the same importance, we compute the weighted individual score:

$$noteCode(c) = note * Imp(c) \quad (2)$$

where $Imp(c)$ is the importance of the entity c given by *Importance algorithm*.

3.2.2. Norms and standards practices

These practices determine compliance with the coding rules. The rules must be respected by the developers. The overall score is the function of the number of transgressions.

There are three types of transgression:

- Errors which are the most heavily weighted, we denote their weight W1;
- Warnings which are moderately weighted, we denote their weight W2;
- Slightly weighted information, we denote their weight W3.

The weight applied to the transgressions reflects the transgression importance, and we also consider the *Imp(c)* importance of the entity where the transgression occurred. It corresponds to the transgression tolerance by several items (methods, classes, etc.). The formula used to calculate the practice grade is of the form:

$$\text{noteStandard}=\mathbf{3} * \frac{\mathbf{2}}{\mathbf{3}}^{\frac{X1*Pe+X2*Pw+X3*Pi}{métric}} \quad (3)$$

Where:

$$Pi, Pe, Pw = \sum_{i} IMP\ (Ci)\ ;$$

$$X1 = W1 * Errors;$$

$$X2 = W2 * Warnings;$$

$$X3 = w3 * \#Infos$$

With W1 the weight applied to *#Errors* (the number of errors), W2 the weight applied to *#Warnings* (the number of warnings), W3 the weight applied to *#Infos* (the number of information), and *metric* the metric used in practice depending on the measured convention. The norms and standard practices are: Formatting, Programming, Naming convention, Configuration management, Tracing, Portability, Documentation, Safety.

3.2.3. Documentary practices

These practices are based on experts' analysis because we cannot evaluate automatically them from metrics. This analysis aims to verify required documents for the project (documentation, user manual, etc.) and to determine their quality. Since these practices are based on human analysis, we do not compute the scores like those based on metrics. Therefore, they also have a lifetime limit and must be recomputed after a time. The documentary practices are: Documentation quality, Exception handling, Technical file: Safety, Layered slicing, Production file, Security implementation/definition compliance, Installation/un-installation manual, Machine independence, System-software independence, Common Communication, Common data. We evaluate these practices using the linguistic terms [3=excellent, 2.5=good, 2=fair, 1.5=average, 0=insufficient]. We represent each evaluated characteristic according to a detailed model. Once the aggregate score is gotten, we convert it into a numerical value in the interval [0; 3] to be consistent with the model.

3.2.4. Testing practices

These practices assess the quality of the tests performed. The base formulas are derived from [17]. Our formulas just take into account the importance of the entity evaluated according to the Algorithm Importance presented in section 3.1. These evaluations, depending on whether they are based on metrics or raw data, following the same logic as the evaluation of code practices. The considered test practices are: Test maturity, Anomaly management, Management of anomalies after tests, Version management, Load tests, Robustness tests, Coverage of unit tests, Coverage of integration tests, Configuration test coverage, Functional tests of non-regression, System integration testing, Configuration/installation tests, Coverage of system integration tests, System data coverage, Success rate of unit tests, Success rate of integration tests, Load test covers, Coverage of non-regression tests, Coverage of robustness tests.

3.2.5. Model practices

The model practices describe the rules and technical principles to be observed to get a good project model. Formulas are derived from [17]. Like testing practices, we evaluate model practices in the same ways as code practices or document practices, depending on whether they are based on metrics or raw data. However, some practices use basic metrics combination to compute their individual scores. We verify specific rules, such as those for accessing attributes, are respected for these practices. For each component, the metric verifies compliance with the rule, and the individual score assigned is

- 0 if the metric returns a negative result, i.e., there is at least one element that violates the rule;
- 3 if the metric returns a positive result, i.e., all measured items follow the rule.

The model practices are: Modeling diagram, Encapsulation, Pre-detection of anti-pattern, Model reasoning, Conformity between modeling and implementation.

3.3. Aggregation of the entity's scores for a practice in order to obtain the score of the practice at the project level

The second step, called aggregation, allows all the results to be grouped to give an overall practice score in the interval [0; 3]. This is a statistical-like operation performed using a weighted continuous function to calculate the overall project score for a practice. All formula are derive from [17].

We can distinguish two cases:

- The score is computed at entities levels; we can aggregate to system level by the general formula for practice mark (for example code practice):

$$PM^{\lambda} = -log_{\lambda}\left(\frac{\sum_{1}^{n}\lambda^{-entityNote_n}}{n}\right) \quad (4)$$

- The score is computed at system level, then formula for practice mark (for example Norm and standard Practice) is:

$$PM^{\lambda} = -log_{\lambda}(\lambda^{-noteP}) \quad (5)$$

Where λ varies according to the applied weight; $entityNote_n$ relates to the individual note for component n and noteP the score for a Practice (example noteStandard above). The weights $\lambda \in \{30,9,3\}$ reflect the tolerance threshold of the model. We define them like this:

- We apply a strong weight λ = 30 when the exception tolerance is extremely low. Here, a few bad individuals mark to get a bad overall mark (included in the interval [0; 1]);
- We use an average weight λ = 9 when the exception tolerance is within the norm. The overall score falls in the interval [0; 1] only if there is an average number of low individual ratings;

We apply a low weight λ=3 in the event of a considerable exception tolerance. The overall rating then only drops if there are many bad individual ratings.

3.4. Aggregation of the project practices weighted scores in order to obtain the maintainability score of the project

At this point we have the notes of each practice for the system. They are aggregated to obtain the system maintainability score using the following formula:

$$OMI^{\lambda} = -log_{\lambda}\left(\frac{\sum_{1}^{n}\lambda^{-PM_n}}{n}\right) \quad (6)$$

4. Evaluation

Before presenting the result of the evaluation, we present the dataset used for. We compare three maintainability indexes: the Maintainability index MI proposed by [9]; the Squale Maintainability Index SMI and Our Maintainability Index OMI.

4.1. Building the dataset

The dataset comprises 20 open-source Pharo system packages publicly available on GitHub *https://github.com/pharo-project*. We judge these projects ideal for our research because Pharo is an open source and immersive nature. In addition, we have access to the Squale model developers. Therefore, we operated on 20 packages. For each package, we evaluate the maintainability according to the Squale model, the classic maintainability index, and our approach. Using the Maintainability Index to study source code and measure maintainability is a proven approach. Although criticized, it remains used by many. This shows the difficulty of unavoidable maintenance tasks. Table 2 summarizes the total number of entities in the dataset (packages, classes, and methods). We selected the final versions of the packages, considering the developers' domain of the projects (packages). Therefore, if the selected projects are from the same developers, they likely have the same development styles. A considerable number of entities must facilitate a comparison of approaches. To run the analysis on the same platform, we selected the Moose tool because it implements most of the building blocks. Our

work will therefore comprise putting together these blocks. Moose is an open source and immersive. It also allows one to do Meta programming and is easily extensible.

The real problem is the made interpretation despite criticisms and shortcomings that must be overcome to make it a widely accepted and representative measure. We experimented on 20 real projects to perform statistical tests. We then submitted our results to 7 systems experts. The results got indicate our approach acceptance.

Table 2. Entities in evaluation dataset

Projects	#line of code	#class	#methods
Balloon	4040	27	584
CodeImport	616	14	140
Colors	2166	3	267
Compression	3327	28	396
Spruce	2229	55	534
Geometry	2139	23	320
Hermes	755	34	203
Iceberg	5956	135	1492
Issue Tracking	56	3	16
Jobs	210	7	50
Merlin	6108	81	821
Monticello	6415	92	1368
Ombu	1156	29	225
PetitParser	2584	46	512
Reflectivity	2324	55	523
Renraku	1810	62	373
Roassal2	95842	775	8533
RoelTyper	1196	13	192
Trachel	7128	125	1306
UIManager	1009	9	235
Total	**147066**	**1616**	**18090**

4.2. Result and discussions

In this section, we summarize the results of the data set evaluation concerning the three approaches. Figure 1 presents the evaluations of maintainability of some packages by the three models. We change the scale of MI to fit 0-3, so that we can visually compare models.

We chose seven experts who have in-depth knowledge of the 20 projects. Each expert has actually contributed to the maintenance of each of the projects through the history of contributions (commits). Each answered the following question: Which of the following measures best represents the maintainability of the system? To each expert, we proposed a graduated notation from 0 (low) to 4 (high) with a step of 0.1. The work consisted in putting its appreciation of the maintainability score of the model of each project in this graduation. The appreciation for a model by an expert is therefore the average obtained from the 20 projects. For the sake of reading, we have multiplied these averages by 5 and validated it by the expert. Table 3 presents the responses. We find that our approach is the most accepted by experts.

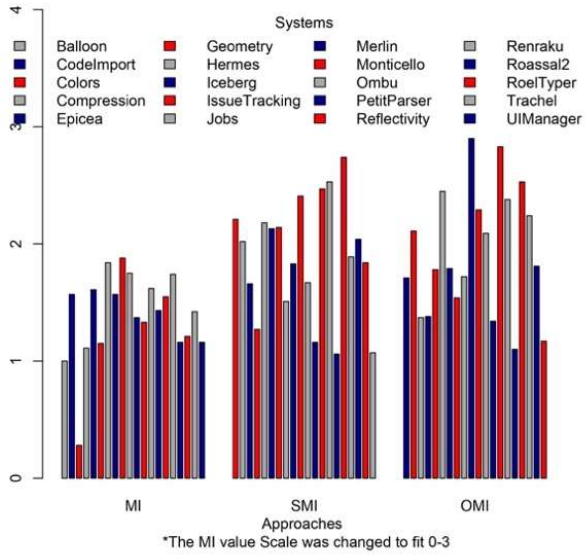

Figure 1. maintainability values per models.

Table 3. Models evaluation by experts

Expert	MI	SMI	OMI
Expert 1	6	8	10
Expert 2	5	11	8
Expert 3	7	7	10
Expert 4	8	10	6
Expert 5	4	12	8
Expert 6	9	8	7
Expert 7	4	4	16
	43	**60**	**65**

Table 4 shows the correlation between the measurements. As we can see, data shows no correlation between models. SMI-MI is the least comparable model, with a correlation=0.3134646. SMI and OMI (correlation=0.1540338) and MI-OMI (correlation=0.1312529) showed similar distributions of correlations, slightly in favor of OMI and MI, which have lower variance.

Table 4. Correlation between models.

	OMI	SMI	MI
OMI	1		
SMI	0.1540338	1	
MI	0.1312529	0.3134646	1

To see if these differences are statistically significant, we run Wilcoxon signed rank tests and paired difference tests to assess mean rank differences for the correlations. The difference is statistically significant for the MI-SMI, SMI-OMI and MI-OMI pairs we have respectively:

- t = 1.3891, df = 18, p-value = 0.1818, 95 percent confidence interval: -0.1523466 0.6624574 sample estimates: cor =0.3111509,
- t = 0.6614, df = 18, p-value = 0.5167 95 percent confidence interval: -0.3095896 0.5584864 sample estimates: cor 0.1540338;
- t = 0.56665, df = 18, p-value = 0.5779 95 percent confidence interval: -0.3294348 0.5430901 sample estimates: cor 0.1323853.

Given the low correlation between MI, SMI, and OMI, we have fewer similarities for the 20 projects chosen. SMI and OMI are the models that show a weaker trend correlation in the 20 projects considered. OMI is also the model that seems more stable compared to other models.

4.3. Threats to validity

Like any empirical evaluation, our experiment's results are subject to validity threats. We have identified the following notable threats:

1. We performed the analysis on open source software systems, all implemented in Smalltalk. The absence of industrial/proprietary systems threatens the general validity of our conclusions. These studied systems may not fully represent a larger population of systems, either systems from another domain or written in another programming language. This threat is still problematic because there is little information about the essential properties for a representative system. Replication of the experiment with other systems must be performed. We strongly believe that our approach is independent of programming language and application domain.

2. Judgments made on non-computable metrics by experts during our experimentation may introduce biases. However, we also believe that the experts who made these judgments are the systems developers, which makes our studies credible.

3. Internal threats to validity are related to the approach implementation. For example, we carried the metrics calculation out based on Moose, a proven open-source tool. The loss of precision is caused by the composition and aggregation of metrics values. Therefore, it is still possible that our implementation of the approach contains errors that may affect the accuracy of our results. To counter this threat, we manually reviewed a subset of the results and found no apparent errors.

5. Conclusion

We have proposed a simple and effective approach for evaluating source code in order to identify and quantify its maintainability. It corrects certain limitations of the traditional models noted in the literature. The approach is functional, exhaustive and takes into account the importance of software entities. It can serve as a guide for directing human inquiry, but one must be aware of the limits of objective measurements. It is adaptable and objectively reflects the level of maintainability of software. The results obtained are easily interpretable.

We compared our approach with SMI and MI on a set of 20 Smalltalk projects. We obtained a weak correlation between MI, SMI and OMI. Thus, each model presents the systems from different angles. The comparison of MI and SMI results is comparable to previous research. These differences presuppose that specialists adopting one or the other model might have different perceptions of the state of the project. Future work will extend the analysis to a larger scale and gain more explanatory insights, such as examining the relationships of models to common software quality metrics. Long-term development of the model to other factors, such as reliability and portability.

Competing Interest Statement

The authors declare no known competing financial interests or personal relationships that could have influenced the work reported in this paper.

Data Availability

No additional data or materials were utilized for the research described in the article.

References

[1] Skyguide, "Swiss Airspace Closed Until Further Notice," Skyguide, 15 06 2022. [Online]. Available: Skyguide. https://www.skyguide.ch/media-centre/post/133592. [Accessed 22 09 2022].

[2] C. Singh, N. Sharma and N. Kumar, "Analysis of software maintenance cost affecting factors and estimation models," *International Journal of Scientific & Technology Research,* vol. 8, no. 9, pp. 276-281, 2019.

[3] B. Seref and O. Tanriover, "Software code maintenability: a litterature review," *International Journal of Software Engineering & Applications (IJSEA),* pp. 69-87, 2016.

[4] J. A. McCall, P. K. Richards and G. F. Walters, "Factors in Software Quality," *NTIS Springfield,,* 1976.

[5] ISO/IEC, "ISO/IEC 9126 - Software engineering – Product quality.," ISO, 2001.

[6] ISO/IEC, "Software engineering Software product Quality Requirements and Evaluation (SQuaRE) Guide to SQuaRE," *ISO Geneva,* 2005.

[7] A. Bergel, S. Denier, S. Ducasse, J. Laval, F. Bellingard, P. Vaillergues, F. Balmas and K. Mordal-Manet, "SQUALE – Software QUALity Enhancement," RMoD Team, INRIA, Lille, France, 2009.

[8] K. Mordal-Manet, "Squash, un modèle d'évaluation des systèmes d'informations," *Université Paris 8,* 2012a.

[9] S. Peter, C. Stanislav and R. Bruno, "Comparing Maintainability Index, SIG Method, and SQALE forTechnical Debt Identification," *Scientific programing,* vol. 2020, no. Hindawi, p. 14, 2020.

[10] K. Mordal-Manet, "Analyse et conception d'un modèle de qualité logiciel," Université Vincennes - Saint-Dénis - Paris 8, Paris, 2012b.

[11] H. Al-Kilidar, K. Cox and B. Kitchenham, " The use and usefulness ofthe ISO/IEC 9126 quality standard," *International Symposium onEmpirical Software Engineering (ISESE 2005),* no. IEEE, p. 126–132., 2005.

[12] H. I., K. T. and J. Visser, " A practical model for measuring maintainability.," *international conference on the quality of information and communications technology ,* pp. 30-39, 2007.

[13] A. Counet, *Amélioration du processus de la maintenance du logiciel par un système informatisé d'aide à la décision,* FUNDP, Namur: Institut d'Informatique, 2007.

[14] M. K. Chawla and I. Chhabra., "SQMMA: Software Quality Model for Maintenability Analysis," *Department of Computer Science & Applications, Panjab University, Chandigarh, India.,* pp. 1-9, 2015.

[15] R. Verma, K. Kumar and H. K. Verma, "Code smell prioritization in object-oriented software systems: A systematic literature review," no. Wiley Online Library, 2023.

[16] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi, D. Poshyvanyk, C. Ghezzi and C. Quing, "An empirical investigation into a large-scale Java open source code repository," *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement,* pp. 1-10, 2010.

[17] Software QUALity Enhancement project, "Squale," 26 Mai 2011. [Online]. Available: https://www.squale.org/index.html. [Accessed 19 Mai 2023].