BLOK II

A Practical Model for Measuring Maintainability

I. Heitlager, T. Kuipers & J. Visser

Proceedings of the 6th International Conference on Quality of Information and Communications Technology, 30-39, 2007
A Practical Model for Measuring Maintainability
– a preliminary report –

Ilja Heitlager
Software Improvement Group
The Netherlands
Email: i.heitlager@sig.nl

Tobias Kuipers
Software Improvement Group
The Netherlands
Email: t.kuipers@sig.nl

Joost Visser
Software Improvement Group
The Netherlands
Email: j.visser@sig.nl

Abstract—The amount of effort needed to maintain a software system is related to the technical quality of the source code of that system. The ISO 9126 model for software product quality recognizes maintainability as one of the 6 main characteristics of software product quality, with adaptability, changeability, stability, and testability as subcharacteristics of maintainability.

Remarkably, ISO 9126 does not provide a consensual set of measures for estimating maintainability on the basis of a system’s source code. On the other hand, the Maintainability Index has been proposed to calculate a single number that expresses the maintainability of a system.

In this paper, we discuss several problems with the MI, and we identify a number of requirements to be fulfilled by a maintainability model to be usable in practice. We sketch a new maintainability model that alleviates most of these problems, and we discuss our experiences with using such as system for IT management consultancy activities.

I. INTRODUCTION

The ISO/IEC 9126 standard [1] describes a model for software product quality that dissects the overall notion of quality into 6 main characteristics\(^1\): functionality, reliability, usability, efficiency, maintainability, and portability. These characteristics are further subdivided into 27 sub-characteristics. Furthermore, the standard provides a consensual inventory of metrics that can be used as indicators of these characteristics [2], [3].

The identification and definition of software quality characteristics of the ISO quality model provides a useful frame of reference and standardized terminology which facilitates communication concerning software quality. The defined metrics provide guidance for a posteriori evaluation of these characteristics based on effort and time spent on activities related to the software product, such as impact analysis, fault correction, or testing. Unfortunately, the listed metrics are not measured on the system itself and lack predictive power. Speaking informally, they predict tomorrow’s weather to be the same as today’s.

In general, the proposed metrics for assessing the maintainability characteristics are not measured on the subject of maintenance, i.e. the system’s source code and documentation, but on the performance of the maintenance activity by the technical staff.

\(^1\)In fact, this subdivision is given for so-called internal and external quality, while a separate subdivision is made for quality in use. See also below.

Many software metrics have been proposed as indicators for software product quality [4], [5]. In particular, Oman et al. proposed the Maintainability Index (MI) [6], [7]: an attempt to objectively determine the maintainability of software systems based upon the status of the source code. The MI is based on measurements the authors performed on a number of systems and calibrating these results with the opinions of the engineers that maintained the systems. The results for the systems examined were plotted, and a fitting function was derived. The resulting fitting function was then promoted to be the Maintainability Index producing function. Subsequently, a small number of improvements were made to the function.

We have used the Maintainability Index in our consultancy practice [8] over the last four years, alongside many other measures, and found a number of problems with it. Although we see a clear use for determining the maintainability of the source code of a system in one (or a few) simple to understand metrics, we have a hard time using the Maintainability Index to the desired effect. A prime reason is that a particular computed value of the MI does not provide clues on what characteristics of maintainability have contributed to that value, nor on what action to take to improve this value.

Based on the limitations of metrics such as the MI, we have formed an understanding of the minimal requirements that must be fulfilled by a practical model of maintainability that is grounded in source code analysis. With these requirements in mind we have started to formulate and apply an alternative maintainability model. In this alternative model, a set of well-chosen source-code measures are mapped onto the sub-characteristics of maintainability according to ISO 9126, following pragmatic mapping and ranking guidelines.

The work reported in this paper is preliminary in the sense that our maintainability model is still evolving. In particular, adjustments and refinements are made to the model on a case by case basis. Nonetheless, the practical value of the model has already been demonstrated in our practise, and we expect further improvements of the model to only bring an increased degree of detail and precision.

This paper is structured as follows. In Section II, we recapitulate the ISO 9126 standard for software product quality, focussing on the characteristics of maintainability. In Section III, we revisit the Maintainability Index and its limitations as perceived by us, which leads up to a formulation in Section IV.
of minimal requirements on a practical maintainability model. The particular model we have come up with is outlined, in simplified form, in Section V. In Section VI, we discuss the merits of the presented model and we sketch its relation to the actual, more elaborate model we employ in practise. We share some experiences and results of the application of this model in our management consulting practise in Section VII. In Section VIII we discuss related work, and we conclude the paper in Section IX.

II. ISO 9126 SOFTWARE ENG. PRODUCT QUALITY

In 1991, an international consensus on terminology for the quality characteristics for software product evaluation was published by the International Standards Organization (ISO): ISO/IEC IS 9126:1991 Software Product Evaluation - Quality Characteristics and Guidelines for Their Use [9]. During the period 2001 to 2004, an expanded version was developed and published by the ISO, which consists of one International Standard (IS) and three Technical Reports (TR):

- IS 9126-1: Quality Model [1]
- TR 9126-3: Internal Metrics [3]
- TR 9126-4: Quality in Use Metrics [10]

The international standard laid down in the first part defines the quality model. The technical reports contain a consensual inventory of measures (metrics) for evaluating the various characteristics defined in the quality models.

A. Views on software product quality

The ISO 9126 quality model distinguishes three different views on software product quality:

- Internal quality: concerns the properties of the system that can be measured without executing it.
- External quality: concerns the properties of the system that can be observed during its execution.
- Quality in use: concerns the properties experienced by its (various types of) users during operation and maintenance of the system.

Internal quality is believed to impact external quality, which in turn impacts quality in use.

B. Characteristics of software product quality

Central to the quality model of ISO 9126 is its breakdown of the notions of internal and external software product quality into 6 main characteristics which are further subdivided into a total of 27 quality subcharacteristics. This breakdown is depicted in Fig. 1. In this paper, we are focussing on the maintainability characteristic, which is subdivided into:

- Analysability: how easy or difficult is it to diagnose the system for deficiencies or to identify the parts that need to be modified?
- Changeability: how easy or difficult is it to make adaptations to the system?
- Stability: how easy or difficult is it to keep the system in a consistent state during modification?
- Testability: how easy or difficult is it to test the system after modification?
- Maintainability conformance: how easy or difficult is it for the system to comply with standards or conventions regarding maintainability?

In the sequel, we will not dwell upon the last of these subcharacteristics.

The third view of quality, i.e. quality in use, is not broken down according to the same hierarchy. Rather, four characteristics of quality in use are distinguished: effectiveness, productivity, safety, and satisfaction. These characteristics are not subdivided further. Quality in use remains out of the scope of this paper.

C. Maintainability measures

Measures for estimating external, internal, and quality-in-use characteristics are listed in three technical reports accompanying the standard quality model. For the maintainability characteristic, 16 external quality measures are defined [2], and 9 internal quality measures [3].

1) External metrics: The suggested external metrics are computed by measuring the performance of the maintenance activity by the technical staff. For example, to measure changeability, the ‘change implementation elapse time’ is suggested. A parameter to this measure is the average time that elapses between the moment of diagnosis and the moment of correction of a deficiency. To measure testability, the ‘re-test efficiency’ is suggested as measure, computed from the time spent to obtain certainty that a deficiency has indeed been corrected. Thus, the maintainability of the software product is estimated by timing the duration of maintenance tasks.

2) Internal metrics: Some suggested internal metrics are based on a comparison of required features and features implemented so far. For example, to measure analysability, the ‘activity recording’ measure is suggested, which is defined as the ratio between the number of data items for which logging is implemented versus the number of data items for which the specifications require logging. Other internal metrics are again based on measurements of the maintenance activity. For example, the ‘change impact’ measure of changeability is computed from the number of modifications made and the number of problems caused by these modifications.

3) Critique: These suggested internal and external measures are not (exclusively) based on direct observation of the software product, but rather on observations of the interaction between the product and its environment: its maintainers, its testers, its administrators; or on comparison of the product with its specification, which itself could be incomplete, out of date, or incorrect.

Therefore, for measuring maintainability by direct observation of a system’s source code, we need to look elsewhere.

III. REVISITING THE MAINTAINABILITY INDEX

The Maintainability Index [6], [7] has been proposed to objectively determine the maintainability of software systems based on the status of the corresponding source code. The MI
The complete fitting function is:

\[ 171 - 5.2 \ln(HV) - 0.23 \text{CC} - 16.2 \ln(LOC) + 50.0 \times \sin(2.46 + \text{COM}) \]

This fitting function is the outcome of data collection on a large number of systems, calibrated with the expert opinions of the technical staff that maintained them. The higher the MI, the more maintainable a system is deemed to be.

In our software quality consultancy practise, we have had the opportunity of calculating the maintainability index for a large and diverse collection of mission-critical software systems. These systems have been developed by many different teams, using many different technologies, for many different purposes. Based on this experience, we have identified a number of important limitations of the MI in the context of software quality assessment.

A. Root-cause analysis

Since the MI is a composite number, it is very hard to determine what causes a particular value for the MI. In fact, since the MI fitting function is based entirely on statistical correlations, there may be no causal relationship at all between the values of ingredient metrics and the value of the MI derived from them. The acceptance of a numerical metric by practitioners, we find, increases greatly when they can determine what change in a system caused a change in the metric. When the MI has a particularly low value, indicating low maintainability, it is not immediately clear what steps can be taken to increase it.

B. Average complexity

One of the metrics used to compose the MI is the average Cyclomatic Complexity. We feel this is a fundamentally flawed number. Particularly for systems built using object-oriented technology, the complexity per module will follow a power law distribution. Hence, the average complexity will invariably be low (e.g. because all setters and getters of a Java system have a complexity of 1), whereas anecdotal evidence suggests that the maintenance problems will occur in the few outliers that have exceptionally high complexity. In general, the use of averaging to aggregate measures on individual system parts tends to mask the presence of high-risk parts.

C. Computability

The Halstead Volume metric, in particular, is difficult to define and to compute. There is no consensual definition of what constitutes an operator or an operand in a language such as Java or C#. For this and other reasons, the Halstead Volume is a metric that is not widely accepted within the software engineering community (e.g. see [13], [14] for a critique). Even if a crisp definition of the notions of operator and operand would be available for all mainstream languages, the Halstead metrics would remain relatively difficult to compute. Basically, a complete and accurate tokenization of all programs needs to be carried out to compute these numbers. For some languages, tokenization is not enough, and a full syntactic and partial semantic analysis is required.

D. Comment

The implication of using the number of lines of comment as a metric is that a well documented piece of code is better to maintain than a piece of code that is not documented at all. Although this appears to be a logical notion, we find that counting the number of lines of comment, in general, has no relation with maintainability whatsoever. More often than not, comment is simply code that has been commented out, and even if it is natural language text it sometimes refers to earlier versions of the code. Also, more documentation for a particular piece of code may have been added, precisely because it is more complex, hence more difficult to maintain. Apparently, the authors of the MI had reservations about measuring comment, as they made this part of the MI optional.
E. Understandability

There is no logical argument why the MI formula contains the particular constants, variables, and symbols that it does. The formula just ‘happens’ to be a good fit to a given data set. As a result the formula is hard to understand and to explain. Why does the formula have two volume measures (HV and LOC) as parameter? Why is the cyclomatic complexity multiplied by 0.23? Why does the count of comment lines appear under a square root and a $\sin$ function? When communicating about maintainability among stakeholders in a system, the recurring invocation of an empirical experimentation as justification for the formula is a source of frustration rather than enlightenment.

F. Control

Using the MI proves to be hard, both on the management level as well as on the technical/developer level. We find that the lack of control the developers feel they have over the value of the MI makes them dismissive of the MI for quality assessment purposes. This directly influences management acceptance of the value. Although having a measure such as the MI at your disposal is obviously more useful than knowing nothing about the state of your systems, the lack of knobs to turn to influence the value makes it less useful as a management tool.

IV. REQUIREMENTS FOR A MAINTAINABILITY MODEL

Based on the limitations of metrics such as the MI, we have formed an understanding of the minimal requirements that must be fulfilled by a practical model of maintainability that is grounded in source code analysis. In particular, we want the following requirements to be met by the various measures to be used in the model:

- Measures should be technology independent as much as possible. As a result, they can be applied to systems that harbour various kinds of languages and architectures.
- Each measure should have a straightforward definition that is easy to compute. Consequently, little up-front investment is needed to perform the measurement.
- Each measure should be simple to understand and explain, also to non-technical staff and management. It should facilitate communication between various stakeholders in the system.
- The measures should enable root-cause analysis. By giving clear clues regarding causative relations between code-level properties and system-level quality, they should provide a basis for action.

In the sequel, we will discuss for each proposed measure whether these requirements are met or not.

V. SIG MAINTAINABILITY MODEL

With these requirements in mind we have started to formulate an alternative maintainability model in which a set of well-chosen source-code measures are mapped onto the sub-characteristics of maintainability according to ISO 9126, following pragmatic mapping and ranking guidelines.

This is by no means a complete and mature model, but work in progress. In fact, the model presented here is actually the stable core of a larger model that has evolved on a case by case basis in the course of several years of software quality consultancy. Evolution has not stopped, and adjustments and refinements are still being made, driven by new situations we encounter, new knowledge we acquire, and retrospective evaluations of each assessment study we perform. In the current paper we share the current state of affairs, and welcome feedback from the academic community.

As illustrated in Fig. 2, the maintainability model we propose links system-level maintainability characteristics to code-level measures in two steps. Firstly, it maps these system-level characteristics to properties on the level of source code, e.g. the changeability characteristic of a system is linked to properties such as complexity. Secondly, for each property one or more source code measures are determined, e.g. source code complexity is measured in terms of cyclomatic complexity. Below we will discuss these two steps in more detail.

A. System characteristics mapped onto source code properties

Our selection of source code properties, and the mapping of system characteristics onto these properties is shown in Fig. 3.

The notion of source code unit plays an important role in various of these properties. By a unit, we mean the smallest...
piece of code that can be executed and tested individually. In Java or C# a unit is a method, in C a unit is a procedure or function. For a language such as COBOL, there is no smaller unit than a program. Further decompositions such as sections or paragraphs are effectively labels, but are not pieces of code that are sufficiently encapsulated to be executed or tested individually.

The influence of the various source code properties on maintainability characteristics of a software system is as follows:

- **Volume**: The overall volume of the source code influences the analysability of the system.
- **Complexity per unit**: The complexity of source code units influences the system’s changeability and its testability.
- **Duplication**: The degree of source code duplication (also called code cloning) influences analysability and changeability.
- **Unit size**: The size of units influences their analysability and testability and therefore of the system as a whole.
- **Unit testing**: The degree of unit testing influences the analysability, stability, and testability of the system.

This list of properties is not intended to be complete, or provide a watertight covering of the various system-level characteristics. Rather, they are intended to provide a minimal, non-controversial estimation of the main causative relationships between code properties and system characteristics. Intentionally, we only highlight the most influential causative links between source code properties and system characteristics. For instance, the absence of a link between volume and testability does not mean the latter is not influenced at all by the former, but rather that the influence is relatively minor.

For ranking, we use the following simple scale for each property and characteristic: ++ / + / o / - / --. We will now discuss the various code-level properties in more detail and for each provide straightforward guidelines for measuring and ranking them.

**B. Volume**

It is fairly intuitive that the total size of a system should feature heavily in any measure of maintainability. A larger system requires, in general, a larger effort to maintain. In particular, higher volume causes lower analysability (the system is harder to understand).

1) **Lines of code**: Many different metrics have been proposed for measuring volume. We could use a simple line of code metric (LOC), which counts all lines of source code that are not comment or blank lines. Within the context of a single programming language, this measure provides sufficient grounds for comparison between systems and for unequivocal rating. For example, a Java system of 200 KLOC could be considered small (+), while a system of 1.3 MLOC or more could be rated as extremely big (--).

2) **Man years via backfiring function points**: However, to meet our requirement that our method be as language independent as possible, we correct for expressivity and productivity of programming languages. For this purpose, we make use of the Programming Languages Table of Software Productivity Research LLC [15]. For an extensive set of programming languages, this table lists (i) how many LOC corresponds on average to a function point (FP), and (ii) how many function points per month a programmer can on average produce when using this language. This leads us to use the following ranking scheme:

<table>
<thead>
<tr>
<th>Rank</th>
<th>MY (Man Years)</th>
<th>Java LOC</th>
<th>Cobol LOC</th>
<th>PL/SQL LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>0 – 8</td>
<td>0-66</td>
<td>0-131</td>
<td>0-46</td>
</tr>
<tr>
<td>+</td>
<td>8 – 30</td>
<td>66-246</td>
<td>131-491</td>
<td>46-173</td>
</tr>
<tr>
<td>o</td>
<td>30 – 80</td>
<td>246-665</td>
<td>491-1,310</td>
<td>173-461</td>
</tr>
<tr>
<td>-</td>
<td>80 – 160</td>
<td>655-1,310</td>
<td>1,310-2,621</td>
<td>461-922</td>
</tr>
<tr>
<td>--</td>
<td>&gt; 160</td>
<td>&gt; 1,310</td>
<td>&gt; 2,621</td>
<td>&gt; 922</td>
</tr>
</tbody>
</table>

Thus, a system larger than 160 man years (MY) is considered extremely large, and is ranked as --. For Java systems, this means that 1.3 million lines of code produce a ++ ranking, while for COBOL this threshold lies only at 2.6 MLOC. When a system consists of programs written in various languages, we simply translate each separate LOC count to man years, add these together, and perform ranking according to the first two columns.

Needless to say, this method of ranking systems by volume is not extremely accurate, but, it has turned out to be sufficiently accurate for our purposes. In fact, we have found our ranking scheme to be highly usable in practise; it is fast, repeatable, sufficiently accurate, explainable, and technology independent.

The requirement of enabling root-cause analysis is not quite fulfilled by the volume measure. When a system is found to be large, the measurement value itself does not immediately indicate causes or possible solutions. By dividing the system into modules, layers, or other partitions, it may be possible to track down those parts that are driving the code bloat. But often, this is not the case, and the excessive volume may simple be the result of attempting to pour too much
functionality into a single system.

3) Other volume measures: Apart from lines of code, or man months calculated via backfiring function points, we frequently use supplementary estimates. For example, for some systems it makes sense to get some estimate of functional size, by counting database tables and fields, screens or input fields, logical and physical files, and such. We employ similar rating schemes in relation to these measures. However, these measures are often not easy to calculate, they are rather language specific, and they do not measure general volume, but functional size specifically. We use them as secondary measures only.

C. Complexity per unit

The complexity property of source code refers to the degree of internal intricacy of the source code units from which it is composed. Complex units are difficult to understand (analyze) and difficult to test, i.e. complexity of a unit negatively impacts the analysability and testability of the system.

1) Cyclomatic complexity per unit: Since the unit is the smallest piece of a system that can be executed and tested individually, it makes sense to calculate the cyclomatic complexity on each unit. As we discussed earlier, the complexity follows a power law distribution, so calculating an average complexity on each unit. As we discussed earlier, the complexity of internal intricacy of the source code units from which it is composed. Complex units are difficult to understand (analyze) and difficult to test, i.e. complexity of a unit negatively impacts the analysability and testability of the system.

2) Other complexity measures: In particular cases, we use complementary measures for complexity. These include structure metrics such as fan-in, fan-out, coupling, and stability measures derived from these. These measures can be computed in many different ways, depending on the interrelations and groupings of units that are taken into account. We have not fixed, language-independent rating schemes for these measures, and we employ them mainly as supplemental to cyclomatic complexity.

D. Duplication

Duplication of source code fragments (code clones) is a phenomenon that occurs in virtually every system. Though the occurrence of a small amount of duplication is natural, excessive amounts of duplication are detrimental to its maintainability, in particular to the characteristics of analysability.
and changeability [18].

Basically, excessive duplication makes a system larger than it needs to be. In fact, we have frequently analyzed systems that were (much) larger than we had expected on the basis of their functionality. We have found that measuring code duplication gives a fairly simple estimate of how much larger a system is. Of course, various other factors also contribute to a system being larger than necessary, including the lack of use of library functions.

Many different techniques have been proposed for finding duplication in source code, also called clone detection [19]–[24]. Most of these techniques have been developed to optimize the trade-off between accuracy, performance, and language independence. We have experimented with several of these sophisticated techniques, but we have settled on an extremely simple method for determining code duplication.

1) Duplicated blocks over 6 lines: We calculate code duplication as the percentage of all code that occurs more than once in equal code blocks of at least 6 lines. When comparing code lines, we ignore leading spaces. So, if a single line is repeated many times, but the lines before and after differ every time, we do not count it as duplicated. If however, a group of 6 lines appears unchanged in more than one place, we count it as duplicated. Apart from removing leading spaces, the duplication we measure is an exact string matching duplication.

Clearly the accuracy of our results is below that of some more sophisticated techniques. However, the duplication measure we defined is again easy to explain and implement, it is fully language independent, and extremely fast. In practise, we have discovered that the accuracy is quite sufficient for our purposes.

Our rating scheme for duplication is as follows:

<table>
<thead>
<tr>
<th>rank</th>
<th>duplication</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>0-3%</td>
</tr>
<tr>
<td>+</td>
<td>3-5%</td>
</tr>
<tr>
<td>o</td>
<td>5-10%</td>
</tr>
<tr>
<td>-</td>
<td>10-20%</td>
</tr>
<tr>
<td>--</td>
<td>20-100%</td>
</tr>
</tbody>
</table>

Thus, a well-designed system should not have more than 5% code duplication. Only exceptionally lean systems show duplication lower than 3%. When duplication exceeds 20%, source code erosion is out of control.

The duplication measure allows root-cause analysis to the extent that the largest duplicates can be listed and the parts of the system where more duplication occurs can be tracked down. However, solving duplication problems often involves more than simply factoring out duplicated fragments into reusable subroutines. Instead, a deeper cause may be present, such as lack of development skills or supporting tools, architectural or design problems, or counter-productive productivity incentives.

E. Unit size

Apart from the complexity per unit, the size of the units from which a system is composed may shed light on its maintainability. Intuitively, larger units are more difficult to maintain because they have lower analysability and lower testability.

As remarked earlier, a strong statistical correlation exists between size (e.g. in terms of LOC) and cyclomatic complexity. The added value of computing unit size in addition to complexity per unit may therefore seem dubious; many of the complex units will also be large. Still, using unit size as a measure complementary to complexity allows detection of large units with low complexity. In our experience, many systems contain a significant number of such units, which should be taken into account when evaluating maintainability.

1) Lines of code per unit: To measure unit size, we again use a simple lines of code metric. The risk categories and scoring guidelines are similar to those for complexity per unit, except that the particular threshold values are different.

F. Unit testing

Unit tests are small programs, written by developers, for automatically testing their code, one unit at a time. For many languages, unit testing frameworks are available that can be integrated into development environments. Examples are JUnit for unit testing of Java code3, and NUnit for .Net languages such as C#4. The presence of an extensive set of good unit tests in a code base has a significant positive impact on maintainability. Unit tests raise testability, since with a single push of a button, tests can be executed. Unit tests raise stability, because they provide a regression suite as safety net to help prevent introducing errors when modifications are made. Unit tests also have a strong documentative nature, which is good for analysability.

1) Unit test coverage: Unit test coverage can be measured with dedicated tools such as Clover4. These tools do not perform static source code analysis, but rather a dynamic analysis which involves running the tests.

Our scoring scheme for unit test coverage is as follows:

<table>
<thead>
<tr>
<th>rank</th>
<th>unit test coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>95-100%</td>
</tr>
<tr>
<td>+</td>
<td>80-95%</td>
</tr>
<tr>
<td>o</td>
<td>60-80%</td>
</tr>
<tr>
<td>-</td>
<td>20-60%</td>
</tr>
<tr>
<td>--</td>
<td>0-20%</td>
</tr>
</tbody>
</table>

Thus, an excellent sets of unit tests covers between 95 and 100% of all production code. A coverage below 60% is considered poor.

The unit test coverage measure does not fulfill our all all requirements. Coverage analysis tools, or even unit testing frameworks, are not available for all languages, and the

1http://www.junit.org/
2http://www.nunit.org/
3http://www.cenqua.com/clover/
The simple measures we have proposed have proven effective in practise. In particular, they allow to reveal significant differences in maintainability even among systems of comparable size, using comparable technologies. This is illustrated in
VIII. RELATED WORK

The usefulness of ISO 9126 for software product quality evaluation has been called into doubt by Al-Kilidar et al. [25]. Their criticism arose from an attempt to apply the standard during an experiment involving pair-design. An important criticism is the following:

“ISO/IEC 9126 provides no guidance, heuristics, rules of thumb, or any other means to show how to trade off measures, how to weight measures or even how to simply collate them.”

We subscribe to this criticism, and the model we presented in this paper is an attempt at provide these missing elements.

Antonellis et al. [26] proposed a method of mapping object-oriented source code metrics onto the maintainability sub-characteristics according to ISO 9126. The metrics are selected from the metrics suite of Chidamber and Kemerer [27]. The method involves elicitation of weights for each pair of metric and sub-characteristic from a system expert. Subsequently, cluster analysis is performed on the calculated results to distribute the units of the system over a fixed number of clusters. The analysis of these clusters provides insight into maintainability problems of the system and their causes. We are currently investigating whether this data-mining approach can complement our model.

Broy et al. [28] have independently developed a similar model of maintainability in which maintenance activities are strictly separated from facts about the system being maintained. Both activities and facts are organized into hierarchical trees whose leaves are related through a (weighted) matrix that indicates which atomic facts influence each atomic activity. The decomposition of activities is based on the IEEE 1219 standard maintenance process, and the decomposition of facts has been developed in concert with industrial partners. A simplified part of the full model is presented only. Specific rating guidelines or weights are not presented.

Oman et al. [29] have proposed a hierarchical structure of measurable maintainability attributes, based on a review of 35 publications. They attach specific software metrics to the leaves of the tree and propose a formula for combining them into a single index. No specific weights are proposed to instantiate that formula. A much larger number of metrics is proposed than in [6], and they include not only source code metrics, but also metrics about e.g. changes, defects found, and documentation.

IX. CONCLUSIONS AND FUTURE WORK

A. Summary

The ISO 9126 standard is a good frame of reference for communication about software product quality, but falls short of providing a practically applicable method of quality assessment. In particular, the metrics listed by the accompanying technical reports can at best establish the degree of maintainability of a system after the fact. The vast literature on software metrics, on the other hand, proposes numerous ways of measuring software without providing a traceable and actionable translation to the multi-faceted notion of quality. In particular, the Maintainability Index suffers from severe
limitations regarding root-cause analysis, ease of computation, language independence, understandability, explainability, and control.

We have argued that a well-chosen selection of measures and guidelines for aggregating and rating them can, in fact, provide a useful bridge between source code metrics and the ISO 9126 quality characteristics. We have presented such a selection, grown out of our software quality assessment practise, that forms the stable core of a practically usable maintainability model. In the course of dozens of software assessment projects performed on business critical industrial software systems, this model has been tested and refined.

B. Future work

Fenton et al. [30, 31] propose the use of Baysian Belief Nets (BBN) in software assessment. Johnson et al. [32] use an extension of BBNs, called influence diagrams, specifically in combination with ISO 9126. The underlying idea is that BBNs capture causal relationships that can not be captured with traditional statistical approaches to software metrics. We would like to investigate how our approach relates to theirs and whether our rating schemas could be captured with BBNs.

The ISO is currently developing the ISO 25000 series (SQuaRE) to complement and partially supersede ISO 9126 [33], [34]. The first parts of this series are expected to be published over the coming two years. We are looking forward to this development, and we hope to incorporate the new standard into our maintainability model.

ACKNOWLEDGMENT

Thanks to Per John, Michel Kroon, and Harro Stokman of the Software Improvement Group for their contributions to the design of the model presented in this paper.

REFERENCES