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 measured *indirectly* 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 directly on the *subject* of maintenance, i.e. the system's source code and documentation, but indirectly 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 by Oman *et al.* 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 a large number of 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 9126-2: External Metrics [2]
- 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 subdived 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:

- Analyzability: 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 sub-characteristics.

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

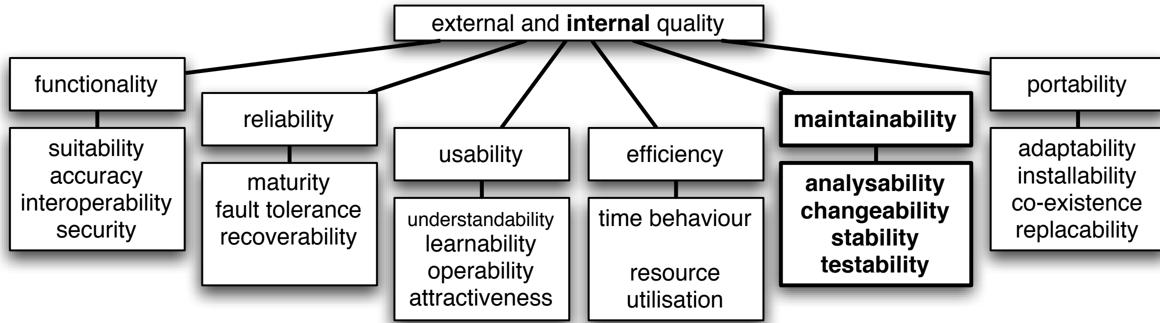

Fig. 1. Breakdown of the notions of internal and external software product quality into 6 main characteristics and 27 sub-characteristics. The 6 so-called compliance sub-characteristics of each of the 6 main characteristics have been suppressed in this picture. In this paper, we focus on the maintainability characteristic and its 4 sub-characteristics of analyzability, changeability, stability, and testability.

based on the status of the corresponding source code. The MI is a composite number, based on several unrelated metrics for a software system. It is based on the Halstead Volume (HV) metric [11], the Cyclomatic Complexity (CC) [12] metric, the average number of lines of code per module (LOC), and optionally the percentage of comment lines per module (COM). Halstead Volume, in turn, is a composite metric based on the number of (distinct) operators and operands in source code. The complete fitting function is:

$$171 - 5.2 ln(\text{HV}) - 0.23\text{CC} - 16.2 ln(\text{LOC}) + 50.0 sin\sqrt{2.46 * \text{COM}}$$

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* relation at all between the values of ingredient metrics and the value of the MI derived from them. The acceptance of a numerical metric with 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 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# [13]. For this and other reasons, the Halstead Volume is a metric that is not widely accepted within the software engineering community (e.g. see [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 stake holders 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 implement and 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 stake holders 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.

	source code properties				
ISO 9126 maintainability	volume	complexity per unit	duplication	unit size	unit testing
analysability	x		x	x	x
changeability		x	x		
stability					x
testability		x		x	x

Fig. 3. Mapping system characteristics onto source code properties. The rows in this matrix represent the 4 maintainability characteristics according to ISO 9126. The columns represent code-level properties, such as *volume*, *complexity*, *duplication*, *unit length*, *number of units*, and *number of modules*. When a particular property is deemed to have a strong influence on a particular characteristic, a cross is drawn in the corresponding cell.

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* of the source code. 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. For a language such as COBOL, there is no smaller unit than a program. Further decompositions such as sections or

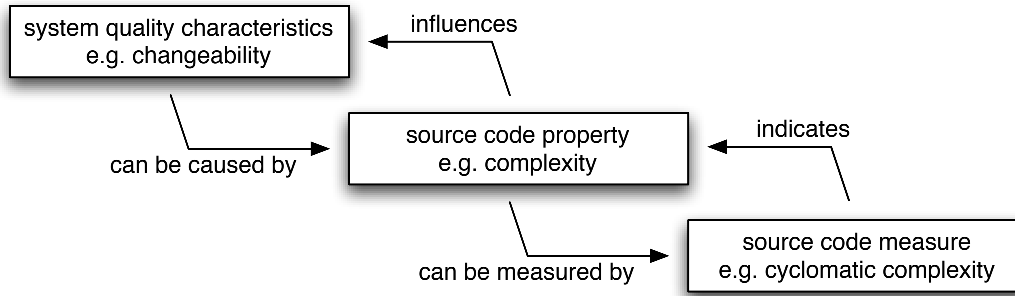

Fig. 2. The maintainability model that we propose maps system-level quality characteristics as defined by the IS 9126-1 standards into source code measures. The first step in this mapping links these system-level characteristics to source code properties. The second step provides a measurement of the properties in terms of one or more source code measures.

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 systems is as follows:

- Volume: The overall volume of the source code influences the analysability of the system.
- Complexity per unit: The complexity of the 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 high-light 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:

		KLOC		
rank	MY	Java	Cobol	PL/SQL
++	$0-8$	0-66	0-131	0-46
+	$8-30$	66-246	131-491	46-173
o	$30-80$	246-665	491-1,310	173-461
-	$80-160$	655-1,310	1,310-2,621	461-922
--	> 160	$> 1,310$	$> 2,621$	> 922

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 poor 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 analyzability 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 of the complexities of individual units will give a result that may smooth out the outliers. Summation of unit complexities provides a complexity number of the entire system. However, this sum has been observed to correlate strongly with volume measures such as total LOC and is, therefore, not meaningful as complexity measure [16]. A different way to aggregate the complexities of units needs to be found.

To arrive at a more meaningful aggregation, we take the following categorization of units by complexity, provided by the Software Engineering Institute, into account [17]:

CC	Risk evaluation
1-10	simple, without much risk
11-20	more complex, moderate risk
21-50	complex, high risk
> 50	untestable, very high risk

Thus, from the cyclomatic complexity of each unit, we can determine its risk level.

We now perform aggregation of complexities per unit by counting for each risk level what percentage of lines of code falls within units categorized at that level. For example, if, in a 10.000 LOC system, the high risk units together amount to 500 LOC, then the aggregate number we compute for that risk category is 5%. Thus, we compute relative volumes of each system to summarize the distribution of lines of code over the various risk levels. These complexity risk ‘footprints’ are illustrated in Fig. 4 for two different systems.

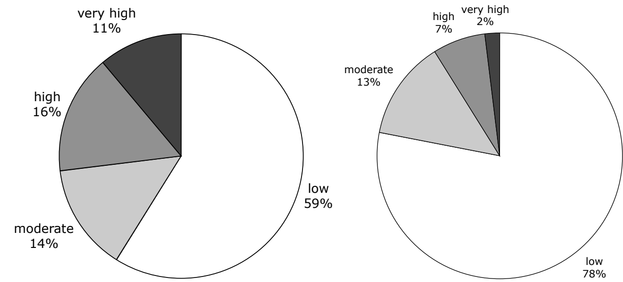

Fig. 4. Distribution of lines of code over the four complexity risk levels for two different systems. Regarding complexity, the leftmost system scores -- and the rightmost system scores -.

Given the complexity risk footprint of a system, we determine its complexity rating using the following schema:

	maximum relative LOC		
rank	moderate	high	very high
++	25%	0%	0%
+	30%	5%	0%
o	40%	10%	0%
-	50%	15%	5%
--	-	-	-

Thus, to be rated as ++, a system can have no more than 25% of code with moderate risk, no code at all with high or very high risk. To be rated as +, the system can have no more than 30% of code with with moderate risk, no more than 5% with high risk, and no code with very high risk. A system that has more than 50% code with moderate risk or more than 15% with high or more than 5% with very high risk is rated as --.

For example, the system with the leftmost complexity profile of Fig. 4 will be rated as –, since it breaks both the 15% boundary for high risk code and the 5% boundary for very high risk code. The rightmost profile leads to a - rating, because it breaks the 0%, but not the 5% boundary for very high risk code.

The boundaries we defined are based on experience. During the course of evaluating numerous systems, these boundaries turned out to partition systems into categories that corresponded to expert opinions. This rating scheme is again language independent, easy to explain and compute, and sufficiently accurate for our purposes. Also, by listing the most complex units, the sources of increase risk and decreased mainainability are easy to track down.

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.

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 [18]–[23]. 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:

rank	duplication
++	0-3%
+	3-5%
o	5-10%
-	10-20%
--	20-100%

Thus, a well-designed system should not have more than 5% code duplication. Only exceptionally lean systems shown 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 analyzability 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 code[2], and NUnit for .Net languages such as C#[3]. 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 Clover[4]. 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:

rank	unit test coverage
++	95-100%
+	80-95%
o	60-80%
-	20-60%
--	0-20%

[2]http://www.junit.org/
[3]http://www.nunit.org/
[4]http://www.cenqua.com/clover/

ISO 9126 maintainability	volume	complexity per unit	duplication	unit size	unit testing	
	++	--	-	-	o	
analysability	x		x	x	x	o
changeability		x	x			-
stability					x	o
testability		x		x	x	-

(source code properties)

Fig. 5. Mapping source code property scores back to system-level scores for maintainability subcharacteristics. A system-level score is derived for each sub-characteristic by taking a weighted average of the scores of relevant (i.e. marked with a cross) code properties. By default, all weights are equal.

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 measure is therefore not language independent. Also, coverage analysis is not trivial to compute, since the analysis is dynamic and requires some degree of tuning for each individual system.

2) Number of `assert` statements: A high level of unit test coverage is easy to obtain by writing unit tests of bad quality. A test that, directly or indirectly, invokes many methods is a 'unit' test only in name, but contributes to a high coverage value. Also, a test that invokes methods, but does not check behaviour (i.e. contains no `assert` statements), contributes to the coverage measure without actually testing anything. Thus, in some situations, the awareness of developers that coverage is being measured may lead to increased coverage without increased 'real' testing. In these situations, it is essential to also measure the quality of unit tests.

To estimate quality of unit tests, we count the number of `assert` statements. This is again a very simple measure, easy to implement, understand, and explain. We currently have no fixed rating scheme in place, but merely use this measure to validate the coverage measure.

G. Source code ratings mapped back to system-level

After scoring individual source code properties, we arrive at a scoring of the sub-characteristics of maintainability by aggregation according to the mapping of Fig. 3. An instance of such a backward mapping of source-code level ratings to system-level ratings is depicted in Fig. 5. Basically, to arrive at a system-level score, a weighted average is computed of each source-level score that is relevant according to the cross marks in the matrix. The weights are all equal by default, but different weighing schemes can be applied when deemed appropriate.

Of course, averaging can be applied again to arrive at a single score for overall maintainability. For the example of Fig. 5, this score would be -, meaning poor maintainability.

We do not attribute much added value to such a single score. Rather, the scores for the various sub-characteristics convey more information, and can be traced back in a straightforward manner to the underlying code-level scores. In contrast to a single number, such as the Maintainability Index, this allows root-cause analysis, serves to point out diverging values, and provides a basis to take steps for improving maintainability.

In the example of Fig. 5, for instance, the poor testability (-) can be traced back to very high complexity (–) and high unit size (-), while unit testing is present, though not complete. Analysability is still average (o), in spite of high complexity (–), because the system volume is rather low (++). To improve maintainability, it would be advisable to refactor highly complex units, which may bring down unit sizes as well.

VI. Discussion

As mentioned in the introduction, the model presented here is only a simplified subset of the more sophisticated model that we actually apply in our consultancy practise. The actual model includes more source code properties, addressing issues such as modularization, architectural compliance, use of frameworks and libraries, and separation of concerns. Also, technical properties that are not purely source-code related are taken into account, concerning for instance the build and deployment process and the employed platform and technology. Some of these involve measuring and rating procedures based on check-lists and associated decision trees.

In both the simplified model presented here and the actual model we employ, all underlying measures are selected to match as much as possible the requirements formulated in Section IV. The measures are easy to calculate and explain. They do not involve obfuscating formulas such as the fitting function of the Maintainability Index. Almost all measures are completely language independent, which guarantees they are applicable to systems that involve different technology mixtures. The understandability of the measures and the traceability of the rating process allows root-cause analysis of maintainability problems and provides a basis for taking corrective action. The use of ISO 9126 as frame of reference implies that the model is grounded in a consensual terminology for software product quality. From discussions with developers of dozens of industrial systems we learn that the measures are well accepted.

Thus, the proposed model does not suffer from the problems identified for the Maintainability Index. It does not generate a single number, so it is not a composite index. It facilitates root cause analysis better than the MI, because it does not use averages. It can be easily explained to both technical personnel as well as to responsible managers. It uses numbers that can be easily influenced by changing the code. Preliminary findings show that these changes in the code make systems more maintainable, according to the maintainers of the systems.

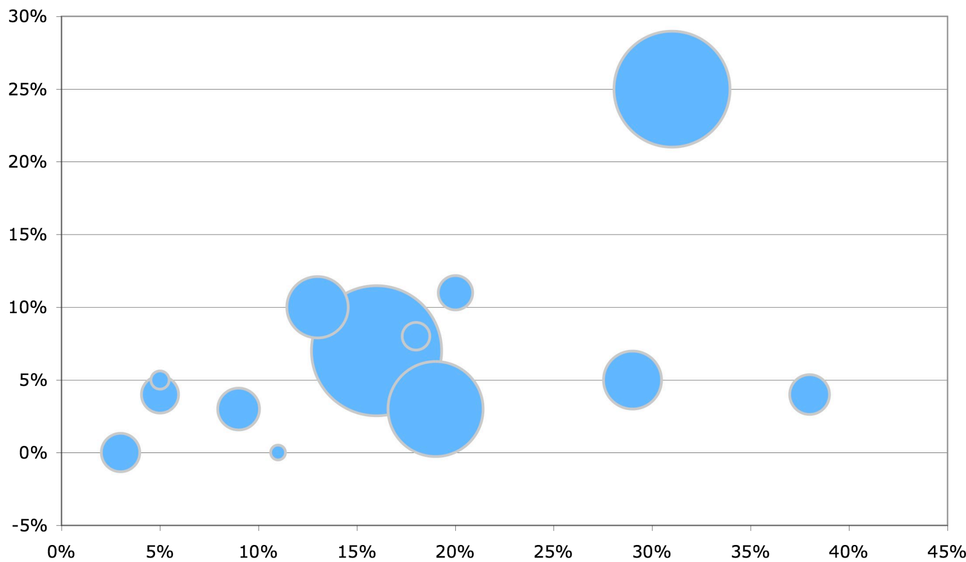

Fig. 6. Three measures applied to a range of modern, object-oriented software systems (Java and C#). The horizontal axis represents degree of duplication. The vertical axis represents percentage of code with cyclomatic complexity above 20. The size of the bubbles indicates volume. Note that systems of comparable size are found to vary significantly in terms of complexity and degree of duplication.

VII. Some measurement results

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 Fig. 6, where volume, complexity, and duplication are plotted for a range of Java and C# systems. For example, for the two biggest systems the degree of duplication varies between 16% and 32%, while the percentage of code with cyclomatic complexity over 20 varies between 7% and 25%. On the other hand, systems with comparable complexity (around 4%) vary in degree of duplication between 3% and 38%.

VIII. Related work

The usefulness of ISO 9126 for software product quality evaluation has been called into doubt by Al-Kilidar *et al* [24]. 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.* [25] 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 [26]. 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.

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.* [27], [28] propose the use of Baysian Belief Nets (BBN) in software assessment. Johnson *et al.* [29] 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 [30], [31]. 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

[1] ISO, "ISO/IEC 9126-1: Software engineering - product quality - part 1: Quality model," Geneva, Switzerland, 2001.

[2] ——, "ISO/IEC TR 9126-2: Software engineering - product quality - part 2: External metrics," Geneva, Switzerland, 2003.

[3] ——, "ISO/IEC TR 9126-3: Software engineering - product quality - part 3: Internal metrics," Geneva, Switzerland, 2003.

[4] N. Fenton and S. Pfleeger, *Software metrics: a rigorous and practical approach*. Boston, MA, USA: PWS Publishing Co., 1997, 2nd edition, revised printing.

[5] H. Zuse, *A Framework of Software Measurement*. Hawthorne, NJ, USA: Walter de Gruyter & Co., 1997.

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

[7] D. M. Coleman, D. Ash, B. Lowther, and P. W. Oman, "Using metrics to evaluate software system maintainability." *IEEE Computer*, vol. 27, no. 8, pp. 44–49, 1994.

[8] A. van Deursen and T. Kuipers, "Source-based software risk assessment," in *ICSM '03: Proceedings of the International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2003, p. 385.

[9] ISO, "ISO/IEC IS 9126: Software product evaluation - quality characteristics and guidelines for their use," Geneva, Switzerland, 1991.

[10] ——, "ISO/IEC TR 9126-4: Software engineering - product quality - part 4: Quality in use metrics," Geneva, Switzerland, 2004.

[11] M. H. Halstead, *Elements of Software Science*, ser. Operating, and Programming Systems. New York, NY: Elsevier, 1977, vol. 7.

[12] T. J. McCabe, "A complexity measure." *IEEE Trans. Software Eng.*, vol. 2, no. 4, pp. 308–320, 1976.

[13] P. A. Szulewski, N. M. Sodano, A. J. Rosner, and J. B. DeWolf, "Automating software design metrics," Rome Air Development Center, Rome, NY, Tech. Rep. RADC-TR-84-27, 1984.

[14] C. Jones, "Software metrics: Good, bad and missing," *Computer*, vol. 27, no. 9, pp. 98–100, 1994.

[15] Software Productivity Research LCC, "Programming Languages Table," Feb. 2006, version 2006b.

[16] M. Shepperd, "A critique of cyclomatic complexity as a software metric," *Softw. Eng. J.*, vol. 3, no. 2, pp. 30–36, 1988.

[17] C. M. Software Engineering Institute, "Cyclomatic complexity – software technology roadmap," http://www.sei.cmu.edu/str/descriptions/cyclomatic.html.

[18] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *WCRE '95: Proceedings of the Second Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 1995, p. 86.

[19] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 1996, p. 244.

[20] K. Kontogiannis, "Evaluation experiments on the detection of programming patterns using software metrics," in *WCRE '97: Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*. Washington, DC, USA: IEEE Computer Society, 1997, p. 44.

[21] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *ICSM '98: Proceedings of the International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 1998, p. 368.

[22] J. Krinke, "Identifying similar code with program dependence graphs," in *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*. Washington, DC, USA: IEEE Computer Society, 2001, p. 301.

[23] S. Ducasse, O. Nierstrasz, and M. Rieger, "On the effectiveness of clone detection by string matching," *J. Softw. Maint. Evol.*, vol. 18, no. 1, pp. 37–58, 2006.

[24] H. Al-Kilidar, K. Cox, and B. Kitchenham, "The use and usefulness of the ISO/IEC 9126 quality standard." in *2005 International Symposium on Empirical Software Engineering (ISESE 2005), 17-18 November 2005, Noosa Heads, Australia*. IEEE, 2005, pp. 126–132.

[25] P. Antonellis, D. Antoniou, Y. Kanellopoulos, C. Makris, E. Theodoridis, C. Tjortjis, and N.Tsirakis, "A data mining methodology for evaluating maintainability according to ISO/IEC-9126 software engineering – product quality standard," in *Special Session on System Quality and Maintainability - SQM2007*, 2007, satellite of CSMR 2007.

[26] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design." *IEEE Trans. Software Eng.*, vol. 20, no. 6, pp. 476–493, 1994.

[27] N. E. Fenton and M. Neil, "Software metrics: roadmap." in *ICSE - Future of SE Track*, 2000, pp. 357–370.

[28] P. Hearty, N. E. Fenton, M. Neil, and P. Cates, "Automated population of causal models for improved software risk assessment." in *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, D. F. Redmiles, T. Ellman, and A. Zisman, Eds. ACM, 2005, pp. 433–434.

[29] P. Johnson, R. Lagerstrm, P. Närman, and M. Simonsson, "System quality analysis with extended influence diagrams," in *Special Session on System Quality and Maintainability - SQM2007*, 2007, satellite of CSMR 2007.

[30] W. Suryn, A. Abran, and A. April, "ISO/IEC SQuaRE. the second generation of standards for software product quality," in *Software Engineering and Applications (SEA 2003)*, M. Hamza, Ed. Acta Press, 2003.

[31] A. Abran, R. Al Qutaish, J. Desharnais, and N. Habra, *ISO-based Model to Measure Software Product Quality*. Institute of Chartered Financial Analysts of India (ICFAI), ICFAI Books, 2007, to appear.