

SBO

SOFTWARE BENCHMARKING

Affordable Software Quality Assessment

Authors	Dr. Lucian Voinea Dr. Hans Sassenburg
Version	1.0
Last update	24 November 2009

Contents

1	Introduction.....	3
2	Process improvement alone is not enough.....	4
3	What is Software Quality?.....	5
	3.1 Maintainability.....	6
	3.2 Reliability.....	9
4	Quality Model Requirements and Design Considerations.....	10
5	Internal Quality Model.....	12
	5.1 Setting reference levels.....	12
	5.2 The IPQI® aggregation formula.....	13
6	External Quality Model.....	16
	6.1 Test levels.....	16
	6.2 Binary Units.....	17
	6.3 The EPQI® aggregation formula.....	18
7	Conclusions.....	23

1 Introduction

In today's dynamic development climate, with its ever-increasing pressure on efficiency and economic payoffs, there needs to be an increased focus on the quality of the products that are being developed. Failure to do so can result in huge losses that may imperil the very existence of the business. The automotive industry is a good example in this respect. The total costs of auto recalls in USA estimated by the NHTSA¹ exceeded \$6 billion in 2004 alone, not including the costs to consumers [Rupp04]. The cumulated losses resulted from large yearly recalls had a substantial contribution to the precarious situation of the automotive industry during the 2008 -2009 economic downturn.

With only a short history behind, software managed to become a ubiquitous aspect of everyday's life. It is not inconceivable to claim that "our civilization runs on software" as Bjarne Stroustrup, the father of the C++ programming language, has very well observed. Consequently, software quality is an essential concern when considering product quality in general. Yet, assessing software quality is still a mostly academic concern with severe implications in the industry and society, where software defects have already caused serious damage and even physical harm [Leveson95]. While defects in financial or text processing programs are annoying and possibly costly, nobody is killed or injured. When software-intensive products fly airplanes, drive automobiles, control air traffic, run factories, or operate power plants, defects can have more serious consequences.

This paper tries to bring software quality assessment closer to the industry. While the basic concepts it communicates have been previously proposed by the research community [Coleman94, Dromey95, Zuse97, Fenton00, Suryn03, Abran07, Heitlager07] and even adopted in industry standards (e.g., the ISO/IEC 9126-1 standard for evaluation of software quality [ISO01]), in this paper the focus lays on "affordability". The question it tries to answer is how to aggregate quality metrics into key performance indicators (KPI) that are easy to obtain and make sense both for software development organizations and end users.

The ultimate goal is to deliver a certification system that would enable a quality-based business model in the software industry. In such model, contracts could be made not only on product functionality but also quality. An independent certification body could serve as arbiter and would contribute to the proliferation of software suppliers that provide good quality products. This would have a beneficial impact not only on industry, but also on society in general.

¹ National Highway Traffic Safety Administration

2 Process improvement alone is not enough

Even if software quality assessment is a less addressed issue, the software industry has put considerable effort in trying to improve the quality of software products. The main focus has been (and still is) on software process improvement, using process models such as CMMI® and anecdotic evidence about its usefulness. Augmenting the evaluation of an organization's process with an analysis of the quality of the software product itself can identify areas where process improvement would result in improved product quality. Furthermore, process improvement should not just focus just on plotting an organization's progress against a model of process maturity. Better product quality (effectiveness) and increased productivity (efficiency) should be the real objectives.

The keys to better software are simply not to be found in process quality alone. Process improvement is an indirect approach to achieve software quality and in spite of reported achievements, remaining questions are:

- How can outsourcing organizations estimate the quality of in-sourced code?
- How can customer-perceived quality be made explicit during product development to software managers, developers and testers?
- How can a software manufacturer know whether a software product is ready for releasing?
- How can potential buyers be objectively informed about the quality of a software product?

To answer these questions, one needs a good understanding of what software quality is and how to assess it.

Next, a model for software quality assessment is presented. The proposed model focuses strictly on the product itself: source code and binary units (i.e., components for which no source code is available). Related artifacts like documentation, as proposed for instance by LaQuSo [2008], are not taken into account. While such artifacts can affect certain quality aspects, their influence is subjective and can be hardly quantified. For example the simple presence of an architecture description document is not by definition a positive aspect when considering product maintainability, especially when the document is out-of-synch with the actual product. Additionally, the model tries to maximize its "affordability" in practice. Therefore it builds on basic software quality indicators that are easy to obtain (i.e., preferably in an automatic way) and that can be aggregated into relevant KPIs.

® CMMI is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

3 What is Software Quality?

In order to understand how quality can be assessed, one needs to have a good understanding of what quality is. According to the ISO/IEC 9126-1 standard [ISO01], there are six characteristics that determine the quality of a software product: functionality, reliability, usability, efficiency, maintainability, and portability. Out of these, reliability and maintainability have a significant implication on what various product stakeholders perceive as software quality. Consequently, the software quality model and assessment methodology that are proposed in this paper are limited to these two characteristics. While this makes the model less complete than the ISO/IEC 9126-1 standard, it greatly simplifies its structure making it more affordable with a minimal loss of relevance.

A pragmatic classification of software quality is given by Steve McConnell [93] in his book, *Code Complete*. McConnell divides quality in two aspects: one that is important for the product users (i.e., external quality) and another that is relevant for its makers (i.e., internal quality). We adopt this classification and give different definitions to the software quality based on a software stakeholder point of view:

External quality is a measure of how well does a software product meet the expectations of its intended users. In most cases this can be interpreted as “absence of defects”. Consequently, this definition of quality focuses on **reliability** as primary characteristic and can be interpreted as "the probability of failure-free operation of a computer program in a specified environment for a specified time" [Musa87].

Internal quality is a measure of how well does a software product support the development and maintenance activities in a project. Since most of the software products being developed today are expected to evolve over time, it is critical that the software also lend itself to modification, thereby reducing the effort needed to rework systems to accommodate new or changed requirements. Consequently, this definition of quality focuses on **maintainability** as primary characteristic, and can be interpreted as the effort required for understanding, modifying and testing a software product.

While the two quality aspects are often interrelated, there is no direct implication between them. While a good internal quality product can create the context for delivering good external quality, this is not sufficient in order to guarantee it. Additionally, products that are very difficult to maintain (i.e., low internal quality) may still be very reliable and perceived as being of good quality by their users. Consequently, one should use the appropriate quality definition for taking decisions affecting one of the two groups of product stakeholders. For example, a business responsible should assess external quality when deciding about releasing a product to the end users, and internal quality when in-sourcing software development/maintenance.

The chosen classification of software quality into internal and external components should not be confused with the one proposed by the ISO/IEC 9126-2 [ISO03-1] and ISO/IEC 9126-3 [ISO03-2] technical reports accompanying the ISO/IEC 9126-1 [ISO01] standard. The internal/external division of quality metrics proposed by these documents does not take a stakeholder point of view, but has a software execution approach. Internal quality metrics are those that can be assessed without executing the software while the external ones are inferred from measurements and observations done on the running software. Additionally, the ISO/IEC classification is orthogonal on the six proposed quality characteristics. For example, the maintainability characteristic has both internal and external components. The stakeholder approach taken in this paper considers the entire maintainability as an internal quality aspect and the reliability as an external one.

One of the previously expressed critiques with respect to the ISO/IEC 9126-1 standard [Heitlager07] is that the proposed software metrics for assessing quality characteristics are not based exclusively on facts extracted from products, but rather on comparisons between products and their specification, or on interactions between products and project teams. Acquiring such metrics requires a very rigorous development discipline and a specialized monitoring framework, both of which are not available in most projects. To ensure a quality model is “affordable” one should aim to base it on software metrics that are readily available or easy to obtain without imposing additional constraints/requirements on the development process. Software metrics that can be extracted automatically from source code, versioning repositories and spreadsheet files are good candidates in this respect.

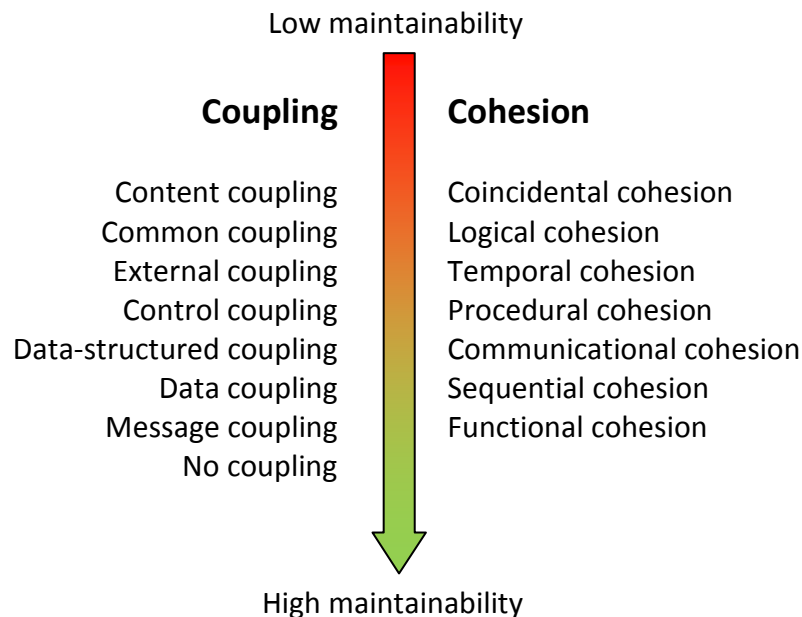
Next, a set of software metrics that meets the “affordability” criteria is presented. This set contains metrics that have been reported to be good indicators of maintainability and reliability, and consequently, are base ingredients of the proposed quality model.

3.1 ***Maintainability***

The most important aspects of maintainability proposed by the ISO/IEC 9126-1 standard [ISO01] are analyzability, changeability, stability and testability. Together, these aspects try to assess how difficult is a software product to understand modify and verify. While the “low coupling, high cohesion” criterion proposed by Constantine *et al.* [1974] stands the test of time, being the cornerstone of all architectural efforts, it is very difficult to assess objectively the degree to which a given product obeys it. Various metrics have been proposed in this respect [Chidamber93, Bieman95, Hitz95, Henderson96]. Most of these metrics are biased towards object-oriented systems, which restricts their usefulness to particular use cases. However, the most important limitation of these measures comes from the difficulty of setting objective criteria in assessing the degree of cohesion a system exhibits.

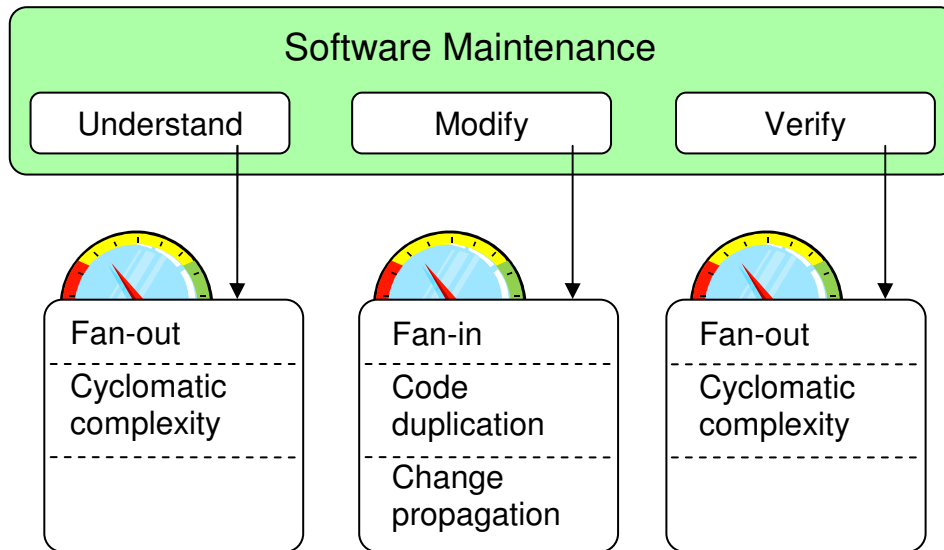
Figure 1 depicts the relation between maintainability and various levels at which coupling and cohesion can be investigated. In general, the lower the level in the image, the higher the maintainability is. However, while the coupling levels form a rather strict precondition stack (e.g., low message coupling requires also low data coupling) the implication between the various cohesion levels is less obvious. Most proposed cohesion metrics (e.g., LCOM [Chidamber93, Hitz95, Henderson96], TCC and LCC [Bieman95]) measure cohesion at the communicational level. Architects, on the other hand, try to achieve functional cohesion as the ultimate target for a maintainable system. This involves very often taking decisions that arguably increase the functional cohesion of a system, yet are reflected in lower communicational cohesion scores. Consequently, current cohesion measurements are not widely accepted in the industry.

Figure 1: Coupling and cohesion relationship to maintainability



In this paper a different approach to assessing maintainability is taken: the main reasons for performing the assessment are used as starting point for selecting a relevant set of software metrics. Consequently, metrics that can quantify how difficult it is to understand, modify, and verify a system are independently selected as ingredients for an integrated maintainability model. These metrics are presented next.

Figure 2: Software metrics for assessing software maintainability



Fan-out is a metric that can be applied on a software component (independent on the definition of component) and gives the number of peers on which the component depends. High fan-out indicates that a component requires the functionality implemented in many other components in order to implement its specification. In general, one should be familiar with the specification of all the required components in order to understand the implementation of an investigated component. Consequently, the higher the fan-out, the more demanding the process becomes. Fan-out is therefore a good indication on how difficult it is to understand software. Additionally, a component with large fan-out can be difficult to test. For these type of components, the implementation of testing scenarios require increased effort, as the behavior of many other components has to be taken into account as well. Therefore, fan-out is also a good indicator for the effort required to thoroughly test a software component.

Fan-in is a similar metric with the fan-out, yet it gives the number of peers that depend on a given software component. A large fan-in is in general a sign of high reuse, and something to be desired in most software architectures with mature interfaces. However, a large fan-in can be dangerous when component interfaces are subject to change. In such cases, a change in the component interfaces is propagated to many other places in the system. Therefore, fan-in is a good indication for the effort required to modify a system with immature interfaces.

Cyclomatic complexity is a static software metric pioneered by McCabe [1976]. The cyclomatic complexity number is a measure of how many linearly independent execution paths are there through a unit of code (independent on the definition of a unit). Previous research [Beizer90] shows that many bugs relate to control flow. Consequently, cyclomatic complexity serves as a good measure of testing effort and as a count of the minimum number of test cases that are required to achieve full base path coverage of the

unit. Additionally, the higher the cyclomatic complexity, the larger the state space of a unit is. This imposes an additional toll on the mental model that a developer makes while trying to understand the workings of a given piece of software. Therefore, cyclomatic complexity can also be an indicator for the effort required to understand a system, albeit to a lesser extent than fan-out.

Code duplication is a measure of how much code has been implemented using “cut & paste development”. This is a very common pattern of code reuse during development. It offers the advantages of structured development without introducing additional abstraction complexity, and therefore can speed up implementation. However, code duplication makes changes in the duplicated code difficult to implement. In many cases changes in one instance of the duplicated code pertain also to the remaining instances. However, because cut&paste development is not traceable, such modifications are not done at the same time, but in a number of development and test iterations, which is an expensive process. Consequently, code duplication gives the probability of having costly modifications of a system.

Change propagation is a measure of how modifications in one part of a software system are likely to propagate to other parts. This measure is based on the patterns of change the software exhibits up to a certain moment. With the advent of configuration management systems, such patterns are recorded in many cases and can be extracted from code versioning repositories. Together with code duplication, change propagation can give an estimation of how difficult it is to modify software.

Figure 2 gives an overview of the proposed metrics and their relation to different maintainability aspects. In Section 5, these metric are aggregated in a model for assessing the internal quality of a software system.

3.2 **Reliability**

Most software reliability estimation models proposed so far (also referred to as reliability growth models) predict future reliability from previously detected faults. Before releasing a product, the input data for such models comes from system testing; afterwards from problem reports in the field. These models attempt to statistically predict failures in the system by fitting defect detection data with known functions, such as an exponential function. The usefulness of such software reliability estimation models has been heavily criticized:

- Most models assume a way of working that does not reflect reality [Whittaker00], meaning that the quality of assumptions is low. As a result, several models can produce dramatically different results for the same data set [Gokhale96].

- Fenton and Neil [1999] studied the most-widely used models and observe many shortcomings. Their study shows that the number of pre-release faults is not a reliable indicator of the number of post-release failures.

In spite of the criticism, many software manufacturers use the pre-release fault count as a measure for the number of post-release failures, and consequently, for the reliability of the released product.

In this paper a pragmatic approach is proposed that uses coded coverage during testing as an indicator for fault occurrence probability. Previous research demonstrates the applicability of this approach in practice [Malaiya98, Bishop02, Cai05]. The idea is that software that has been thoroughly tested is less likely to fail during operation. While this approach is not a good indicator for the amount of residual faults, it communicates the probability of faults causing failures, which is very close to the considered definition of reliability [Musa87].

The main criticism regarding this approach addresses the lack of appropriate means to measure code coverage without interfering with the analyzed product. However, with the advance of profiling and code coverage analysis tools, the proposed approach is gaining increased popularity.

4 Quality Model Requirements and Design Considerations

Besides easy access to raw information, an affordable model for quality assessment has to meet a number of requirements that ensure its usefulness in practice. These are: ease of use, ease of computation, controllability, traceability, and perception fidelity.

The model should be **easy to use** as a key performance indicator (KPI) for the quality of a software product. To this end, the model aggregates a number of metrics into one figure that facilitates monitoring, tracking and communication. Additionally, the figure is normalized, to abstract from the specific product context. This enables cross-project comparisons and the definition of standard quality levels.

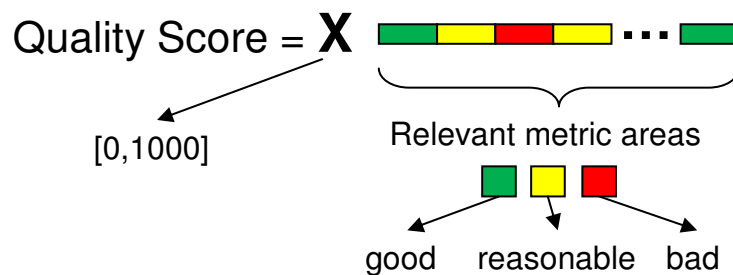
To have practical use, a model for software quality should be **easy to aggregate** from raw data. A useful model should mimic in its structure the software development scheme. That is, the model should enable assessing the quality of higher level software components by **composing** the quality scores of the involved components at lower levels. This would enable the assessment and monitoring of quality at different levels in the organization and at different development phases.

The model results should be **controllable**. When acting on the root cause of a problem expressed by a low quality score, one should notice improvement during subsequent quality assessments. To this end, the score associated with the proposed quality model should have sufficient resolution to reflect even relatively small improvements. The

interval [0, 1000] has been chosen as a compromise between resolution and ease of comprehension.

To enable acting on issues that cause low quality scores, these have to be **traceable**. While a single normative figure is easy to monitor and to use in management communications, it makes it difficult to locate the root cause of problems. To address this drawback, the quality score is enriched with a map that gives a “traffic light” style assessment of the metric areas considered in the model (see Figure 3). The map can be implemented using colored icons or letter suffixes reflecting the corresponding level of each metric area of the model.

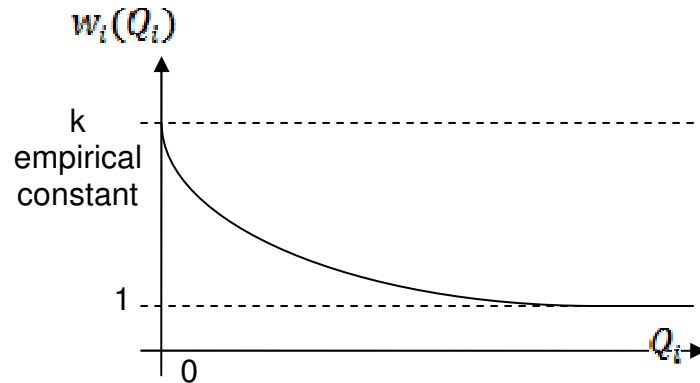
Figure 3: Quality score format



To be useful and easily accepted in practice, a quality model should be highly connected with the **perceived** value it communicates to the stakeholder. For example, a car in which everything but the breaks works fine, communicates however a low value to the end user, and that should be reflected in the model as well. To this end, the proposed quality model introduces a bias towards the “weakest link”. This bias is implemented by using weighted averages when computing aggregate figures out of basic metric results. The weights in the formulas are metric dependent and favor weak/low quality giving numbers. Figure 4 illustrates the proposed principle of aggregation.

Figure 4: Weighted average computation based towards the "weakest link"

$$\text{Aggregated Quality} = Q_{\text{aggregated}} = \frac{\sum_1^n w_i \cdot Q_i}{\sum_1^n w_i}$$



Next, the actual models for external and internal quality are defined by aggregating the basic software metric ingredients identified in Section 3 using the design principles set forth in Section 4.

5 Internal Quality Model

In this Section, a model for the internal quality of a software system is proposed. This model aggregates the software maintainability measures introduced in Section 3 in a single normative figure: the Internal Product Quality Index (**IPQI**[®]). This figure aims to make software quality transparent to software managers and business responsables in order to enable informed decision making regarding software acquisitions, project planning and associated risks.

Next, the normalization issues with the internal quality model are discussed and put in the context of specific organizations. Subsequently, the overall formula and procedure for calculating **IPQI**[®] is provided, together with an illustrative example.

5.1 Setting reference levels

One of the requirements on the quality model proposed in this paper is to enable cross project comparisons and the definition of standard quality levels. This requirement, however, has to be interpreted from the point of view of the stakeholder in the quality model. In the case of internal software quality, the stakeholder can be a development team, a project, or even an entire organization. In either case, the organizational context marks the boundaries for the relevance of normalization values for raw measurements. Cross-organizational comparisons of these values are not meaningful. That is, a software stack that has good internal quality for one development team (i.e., based on certain

raw measurements) may be perceived differently by another team. Two questions arise from this:

- How to normalize quality measurements to enable cross-project comparisons inside the organizational context?
- What is the use of an internal quality model at levels above the one in which it is computed from raw measurements?

Normalization of raw quality measurements is important for enabling cross-project comparisons inside the chosen organizational level. Therefore, all aspects that are specific to the organizational level (but not to the project) can be used in setting normalization values. In practice this takes the form of an iterative and continuously evolving process of adjusting references to the match the current situation. At any moment, an organization should have a reference level for what is considered as being of good internal quality. When taking decisions based on the product quality, results of raw quality measurements are normalized against the reference values, and clamped to the [0,1000] interval if needed. This enables organizations to benchmark current and unknown situations against past and proven experiences, and therefore supports informed decision making.

A second important issue with internal software quality is its usefulness outside the context in which it is computed. While reference values cannot be used for normalization in peer organizations (i.e., “what is good for me is not necessarily good for you”) the normalized quality results can be compared and aggregated at higher levels using the biased composition scheme described in Section 4. The higher levels would be raw measurement agnostic, yet they would still be able to assess, monitor and trace internal quality.

Next section gives the proposed formulas for computing internal software quality from raw measurements and for aggregating quality at higher levels.

5.2 *The IPQI[®] aggregation formula*

The proposed formula for computing IPQI[®] from raw measurements is based on the maintainability indicators presented in Section 3.1. The formula has three components: Q_{MU} – the understandability quality, Q_{MM} – the modifiability quality, and Q_{MT} – the testability quality.

$$IPQI_{raw} = \frac{w_1 \cdot Q_{MU} + w_2 \cdot Q_{MM} + w_3 \cdot Q_{MT}}{w_1 + w_2 + w_3} \cdot 1000 \quad (5.1)$$

where:

$$\bullet \quad Q_{MV} = k_1 \cdot \frac{M_{fan-out}}{R_{fan-out}} + k_2 \cdot \frac{M_{complexity}}{R_{complexity}} \quad (5.2)$$

$$\bullet \quad Q_{MV} = k_3 \cdot \frac{M_{duplication}}{R_{duplication}} + k_4 \cdot \frac{M_{change}}{R_{change}} + k_5 \cdot \frac{M_{fan-in}}{R_{fan-in}} \quad (5.3)$$

$$\bullet \quad Q_{MV} = k_6 \cdot \frac{M_{fan-out}}{R_{fan-out}} + k_7 \cdot \frac{M_{complexity}}{R_{complexity}} \quad (5.4)$$

with:

- $M_{fan-out}$, $R_{fan-out}$ = the current/reference average fan-out;
- $M_{complexity}$, $R_{complexity}$ = the current/reference average cyclomatic complexity;
- $M_{duplication}$, $R_{duplication}$ = current/reference duplication;
- M_{change} , R_{change} = current/reference change propagation;
- M_{fan-in} , R_{fan-in} = the current/reference average fan-out;
- $k_1 = 75\%$, $k_2 = 25\%$, $k_3 = 30\%$, $k_4 = 60\%$, $k_5 = 10\%$, $k_6 = 50\%$, $k_7 = 50\%$ (empirical constants);
- w_1, w_2, w_3 = biased weights (see Section 4).

The formula for estimating internal quality at higher levels in the organization, abstracting from raw measurements is:

$$IPQI_{higher} = \frac{\sum_{i=1}^{ND} wD_i \cdot IPQI_{lower}}{\sum_{i=1}^{ND} wD_i} \quad (5.5)$$

where:

- $IPQI_{lower}$ is the **IPQI**® on the lower level (e.g., $IPQI_{raw}$ for the lowest level);
- wD_i is a weighting factor that favors low **IPQI**® scores (see Section 4);
- ND is the total number of departments reporting $IPQI_{lower}$.

The **IPQI**® values computed using formulas (5.1) or (5.5) are normative values in the interval [0,1000] that give only the large picture. In order to facilitate traceability, color or letter suffixes have to be added to these numbers indicating how the final value has been obtained. For $IPQI_{raw}$ a three letter/color suffix could indicate the values of Q_{MU} , Q_{MM} , and Q_{MT} while for the $IPQI_{higher}$ a custom suffix depending on the total number of departments can be provided.

Example

Consider an organization with two software departments (A and B), each department being characterized by a set of reference values for the software fan-out, fan-in, cyclomatic complexity, code duplication, and change propagation that it can handle. Assume the current measurements given in Table 5.1 and a biasing weight profile given by the function:

$$w(q) = \frac{1000}{100+q}, \text{ with } q \in [0,1000] \text{ a normalized value} \quad (5.6)$$

Dpt.	Fan-out		Fan-in		Cyclomatic complexity		Code duplication		Change propagation	
	R	M	R	M	R	M	R	M	R	M
A	4	6	5	5	5	7	5%	10%	12	18
B	6	6	6	5	6	7	10%	9%	20	18

Table 5.1: Example current and reference values for raw maintainability measurements

In order to compute $IPQI_{raw}$ for department A, one needs to find the values for Q_{MU} , Q_{MM} , and Q_{MT} :

$$Q_{MU_A} = 0.75 * (4/6) + 0.25 * (5/7) = 0.5 + 0.094 = 0.594$$

$$Q_{MM_A} = 0.3 * (5/10) + 0.6 * (12/18) + 0.1 * (5/5) = 0.15 + 0.4 + 0.1 = 0.65$$

$$Q_{MT_A} = 0.5 * (4/6) + 0.5 * (5/7) = 0.333 + 0.357 = 0.69$$

$$IPQI_{raw_A} = (1.441 * 0.594 + 1.333 * 0.65 + 1.265 * 0.69) * 1000 / (1.441 + 1.333 + 1.265) = 642$$

In a similar way:

$$Q_{MU_B} = 0.75 * (6/6) + 0.25 * (6/7) = 0.75 + 0.214 = 0.964$$

$$Q_{MM_B} = 0.3 * (10/9) + 0.6 * (20/18) + 0.1 * (6/5) = 0.3 + 0.6 + 0.1 = 1$$

$$Q_{MT_B} = 0.5 * (6/6) + 0.5 * (6/7) = 0.5 + 0.428 = 0.928$$

$$IPQI_{raw_B} = (0.94 * 0.964 + 0.91 * 1 + 0.973 * 0.928) * 1000 / (0.94 + 0.964 + 0.973) = 945$$

The overall $IPQI^{\circledR}$ of the organization is:

$$IPQI_{overall} = (1.348 * 642 + 0.957 * 945) / (1.348 + 0.957) = 768$$

Consider the letter/color encoding of quality levels given in table 5.2.

Letter	Interval	Certification level
A	801 – 1000	Excellent
B	601 – 800	Good
C	401 – 600	Fair
D	201 – 400	Poor
E	0 – 200	Very poor

Table 5.2: Letter and color encoding of quality levels

The $IPQI^{\circledR}$ assessments relevant for the organization will be:

$IPQI_{raw_A} = 642$ CBB → The current project in department A can be handled quite well; delays are not likely; the major problem the project faces is the difficulty to understand previously written code. Improvement effort should be focused on that aspect (e.g., update the documentation, acquire tools to facilitate code browsing, etc.).

$IPQI_{raw_B} = 945$ AAA → The current project in department B is handled very well; delays are excluded; there is a very high probability the project will be ready within time and budget.

$IPQI_{overall} = 768$ BA → The overall situation in the organization is good. Improvement efforts should be focused on department A.

6 External Quality Model

In this Section, a model for the external quality of a system is proposed. This model aggregates code coverage measurements obtained during testing in a single normative figure: the External Product Quality Index (**EPQI**[®]). This figure aims to make software quality transparent to the end users, and to enable development organizations to take informed decisions regarding product releasing and associated risks.

Next, a number of aspects related to the affordability of code coverage measurements during testing are discussed, and the formula for computing **EPQI**[®] is given. A small example is also provided, to illustrate the **EPQI**[®]-based quality assessment procedure.

6.1 Test levels

A software product normally consists of different units that are integrated. To predict the quality of the resulting software product, three testing levels could be taken into account:

- Unit level. It is important to know how well each unit is coded. This can be assessed by collecting and analyzing data resulting from unit testing, understanding that test coverage is a good indicator for reliability.
- Integration level. It is important to know how well each unit integrates with its interfacing units. This can be assessed by collecting and analyzing data resulting from integration testing.
- System level. It is important to know how well an integrated software product behaves in its operational environment. However, testing code is different from testing a system. The focus of system testing is to verify that all defined functionality has been implemented (“Have we built the product right?”) and to validate that the final product meets the demands of its intended customers/end-users (“Have we built the right product?”). Although system testing can be used to detect defects in the code as well, there are severe limitations:
 - Detecting and removing defects during system testing is difficult, as the possible input combinations thousands of users can make across a given

software interface are simply too numerous for testers to apply them all [Whittaker00]. Code that works with one input will break with another. The number of tests required achieving any given level of test coverage during system testing increases exponentially with software size and therefore becomes impractical.³

- Another complicating factor stems from the dynamic nature of software [Harrold99]. If a failure occurs during testing software version n and the code is changed, the new software version $n+1$ may now work for the test case(s) where it didn't previously work. But its behaviour on other pre-error test cases, previously passed, is not necessarily guaranteed. Any specific fix can (a) fix only the problem reported, (b) fail to fix the problem, (c) fix the problem but damage something that was previously working, or (d) fail to fix the problem and damage something else [Whittaker00]. To account for this possibility, testing should be restarted. However, the incurred costs of re-executing all previous test cases are often prohibitive.

Consequently the system level is not considered in the proposed model or external quality. The scope of the model is restricted to unit and integration testing. Put differently, in order to be affordable, a quality model should focus on those phases of product development where quality levels are established and where there is still room for quality improvements.

In conclusion, a requirement for obtaining high product quality in terms of reliability is to ensure it from the very beginning and on the building blocks of the software product. The quality of large programs depends on the quality of the smaller programs of which they are built. Thus, to produce high quality large software products, every single unit must be of high quality.

Consequently, the proposed model for external software quality enables one to define a start criterion for system verification and validation ensuring that the underlying units have sufficient quality and can be integrated.

6.2 *Binary Units*

As software development has grown more complex, it has become a multi-tier process with many parties involved. Software products are now made of code developed by the

³ Consider, for example, a well-known program which finds the greatest common divisor of two positive integers, with an upper limit of 1,000 on the input values. To exhaustively test this program it is necessary to choose every combination of the two inputs, leading to [1,000 x 1,000] possible distinct input combinations. If the program is modified to allow input in the range 1 to 1,000,000, and include a further input [so it is now finding the greatest common divisor of three numbers] then the input domain grows to [1,000,000 x 1,000,000 x 1,000,000] distinct input combinations. If it was possible to create, run, and check the output of each test in one-hundredth of a second, then it would take over 317 million years to exhaustively test this program.

software manufacturer itself, open source code, code from sub-contractors, offshore providers and third-party components. Lower up-front costs, and a belief that the cost savings extend throughout the software product's life cycle are primary motivators in the shift from fully custom-designed to integrated products.

One of the disadvantages associated with most integrated third-party software is the absence of source code (i.e., the software is available in binary format only). The burden of minimizing risk and controlling operational cost from low-quality binary units is placed largely on the integrators [Basili01]. However, in most cases organizations do not have any insight into what vulnerabilities exist in the used binary units, resulting in an unacceptable level of unbounded risk. White-box testing such units before product deployment is nearly impossible due to the lack of access to source code (intellectual property). As a result, organizations either do not test binary units at all, or perform a shallow investigation of a small sub-set of the purchased software via black box testing, which results in unbounded risks to the organization.

In conclusion, the quality of the integrated binary units influences the quality of the overall software product. Therefore, a model for external software quality should take into account the quality contribution of each binary unit. In order to be affordable, this has to be done in a way that doesn't require vendors to expose any of their intellectual property (e.g., in the form of source code). Consequently, the proposed model requires vendors to deliver the binary units together with an indication of their external quality. This indication has to be obtained via an assessment done with the same model.

6.3 *The EPQI[®] aggregation formula*

The proposed model for external quality has three perspectives on a given product: the unit testing perspective, integration testing and binary unit perspective (see Section 6.1 and 6.2). In each of these perspectives, quality aspects are assessed based on code coverage measurements (see Section 3.2).

In the unit testing perspective, a biased branch-based code coverage measure (BC) is used. The bias is provided by a statement coverage measure (SC). This ensures that both branches and statements are sufficiently covered in order to obtain high quality. While any branch-based coverage can be used, the modified condition decision coverage measure (MCDC) is preferred, as this is already part of the *DO-178B Level A aviation* standard [RTCA92]. The formula for the total external quality from the unit testing perspective (QU) is:

$$QU = \frac{\sum_{i=1}^{NC} wU_i \cdot QU_i}{\sum_{i=1}^{NC} wU_i} \cdot 1000 \quad (6.1)$$

where:

- $QU_i = MCDC_i \cdot SC_i \in [0,1]$ is the external quality of unit i from the unit testing perspective;

- $MCDC_i \in [0,1]$ is the modified condition decision coverage of a software unit i ;
- $SC_i \in [0,1]$ is the statement coverage of software unit i ;
- wU_i is a weighting factor that favors low coverage scores (see Section 4);
- NC is the total number of source code units in the product.

The formula gives total freedom in choosing one's own definition of a unit. The smaller the amount of code that one names a unit, the higher the reported quality values should be. This means that if the quality of a product is not high enough, one will be encouraged to break it in smaller pieces when trying to improve the reported figures. However, by doing so, one will also increase the code coverage of the testing, and consequently the probability of finding bugs. Therefore the quality will also increase in a natural way. On the other hand, if the measured quality is very good, one can choose as higher levels of abstraction for the unit definition, which simplifies the quality assessment procedure.

In the integration testing perspective, the possible interactions of a software unit with its peers are considered. In a high quality product all possible interactions should be tested. Not only individual function calls across units are important in this respect, but also the state of the involved units when the call has been made. In general, it is not practical or even possible to assess whether all interactions have been tested, even with a white box testing approach. One approximation for assuming that all possible states of a unit have been considered during integration testing is to aim for full coverage using the same coverage measure as in the unit testing perspective. However, even for a high coverage value, it is possible that the very state values that are relevant for interaction have not been considered during testing. Similarly, it is possible to address all relevant values with a very small coverage value. In both cases, large inconsistencies can appear between the measured coverage and the actual quality of the integration.

To focus the coverage approximation of relevant states for interaction, the coverage measure can be combined with a data flow analysis. The data flow analysis should provide all unit functions that are relevant for making a function call across units. For example, the data flow analysis should give all functions that contribute directly to setting the value of the parameters that are used in the function call. A similar reasoning holds for functions being called by other units; in such case the data flow analysis should indicate places where variables depending on the called function parameters are used. The coverage analysis should be then applied only on the identified functions.

Using the proposed approach, the quality Q_i of a software unit i from the perspective of integration testing can be then computed as a weighted sum of all relevant integrations j for the given unit i :

$$QI_i = \frac{\sum_{j=1}^{M(i)} wIU_j \cdot QU_j}{\sum_{j=1}^{M(i)} wIU_j} \quad (6.2)$$

where:

- $QU_j = MCDC_j \cdot SC_j \in [0,1]$ is the biased branch coverage measure used in the unit testing perspective but applied only on the code identified as relevant to the considered integration (i.e., via data flow analysis);
- wU_j is a weighting factor that favors low coverage scores (see Section 4);
- $M(i)$ is the number of integrations relevant to unit i ;
- A relevant integration for unit i is a tuple (U,m) with U another unit and m a public member in the interface of unit i and used by U , or in the interface of U and used by i .

The total formula for assessing external quality of a product from the perspective of integration testing is:

$$QI = \frac{\sum_{i=1}^{NC} wI_i \cdot QI_i}{\sum_{i=1}^{NC} wI_i} \cdot 1000 \quad (6.3)$$

where:

- $QI_i \in [0,1]$ is give above;
- wI_i is a weighting factor that favors low coverage scores (see Section 4);
- NC is the total number of source code units in the product.

From the binary unit perspective, it is not possible / practical to reliably assess the code quality of a binary code using a black box testing approach only. Therefore, the proposed model requires that the quality of such units is assessed by their developers using a white box testing approach at development time, and passed further together with the binary unit. The external quality should be assessed by developers using the integrated **EPQI**[®] formula (5) from the unit and integration testing perspectives only. The proposed formula for the total external quality from the binary unit perspective is:

$$QB = \frac{\sum_{i=1}^{NB} wB_i \cdot EPQI_i}{\sum_{i=1}^{NB} wB_i} \quad (6.4)$$

where:

- $EPQI_i \in [0,1]$ is the external product quality index provided by the manufacturer of a binary unit i ;
- wB_i is a weighting factor that favors low coverage scores (see Section 4);
- NB is the total number of binary units in the product.

The overall **EPQI**[®] formula for the external quality assessment that integrates the three considered perspectives is:

$$EPQI = \frac{\sum_{i=1}^N wO_i \cdot EPQI_i}{\sum_{i=1}^N wO_i} \quad (6.5)$$

where:

$$EPQI_i = \begin{cases} \sqrt{QU_i \cdot QI_i} \cdot 1000 & | \text{source code units} \\ \text{reported EPQI} & | \text{binary objects} \end{cases} \quad (6.6)$$

and wO_i is a weighting factor that favors low $EPQI_i$ scores (see Section 4).

The **EPQI**[®] calculated using formula (5.6) is a normative figure between 0 and 1000 that characterizes the overall external code quality of a software product. However suitable for giving the big image on the quality of a software product, a single number cannot reveal the particular combination of the three quality perspectives mentioned above. This would be useful to know in order to understand the type of risk associated with a given **EPQI**[®]. For example, the same **EPQI**[®] could be constructed from high quality source code and low quality binary units or vice versa. Even if the index is identical in the two cases, the conclusion can be very different. Suppose a software manufacturer considers buying a low **EPQI**[®] third-party party software stack. If the low value is caused by binary units that can be easily replaced with high quality ones, the manufacturer might go on with the purchase. If, however, the quality of the source code level is low, the purchase would probably be declined.

Therefore, using the design considerations given in Section 4, **EPQI**[®] is presented together with a three letter pattern that presents the individual contribution of each of the three quality perspectives considered. By requiring that all three individual aspects are expressed on the same scale between 0 and 1000, a range can be defined [A (Excellent) – E (Very poor)] for both the **EPQI**[®] and the quality of each aspect. See Table 6.1.

Letter	Interval	Certification level	Typical Industry Segment ⁴
A	801 – 1000	Excellent	Aerospace, Medical
B	601 – 800	Good	Telecom
C	401 – 600	Fair	Finance, Automotive
D	201 – 400	Poor	Entertainment
E	0 – 200	Very poor	-

Table 6.1: Mapping between letters, quality interval and rating.

⁴ These are assumptions at this stage. Further research towards the requirements as well as industry averages for each industry segment is needed.

The three letter pattern indicates the quality of the associated aspect: the first letter is reserved for unit testing, the second letter for integration testing and the last letter for binary components. For example the $EPQI = 750 ABC$ indicates:

- The software product exists of source code units of excellent quality (A);
- These units are well integrated (B);
- The binary units have fair quality (C).

Example

Consider a software product, consisting of two units and two binary units, with the characteristics as in Table 6.2:

Unit name	Type	Unit Quality (QU_i)	Integration Quality (QI_i)	Reported Quality ($EPQI_i$)	Quality bias (wU)	Quality bias (wI)	Quality bias (wB)	Quality bias (wO)
A	Src	0.1	0.6		10	2		7
B	Src	0.2	0.9		8	1		4
C	Binary			700	1		1.5	1.5
D	Binary			800	1		1	1

Table 6.2: Software product characteristics.

We can compute the following indexes:

$$\begin{aligned}
 QU &= (10/18 * 0.1 + 8/18 * 0.2) * 1000 &= 144.44 &(E) \\
 QI &= (2 * 0.6/3 + 0.9/3) * 1000 &= 700 &(B) \\
 QB &= 1.5/2.5 * 700 + 1/2.5 * 800 &= 740 &(B)
 \end{aligned}$$

This leads to:

$$\begin{aligned}
 EPQI &= 7/13.5 * 0.24 * 1000 + \\
 &4/13.5 * 0.42 * 1000 + \\
 &1.5/13.5 * 700 + \\
 &1/13.5 * 800 &= 386
 \end{aligned}$$

So, the overall **EPQI**[®] equals 386 EBB. One can infer from this that the external quality of the system is poor. While the quality of the binary units, and of the unit integration is good, the quality of unit testing is very poor and any improvement effort should be concentrated on that aspect.

7 Conclusions

This paper presents a model for assessing the quality of a software system using a stakeholder approach. To this end software quality is divided into internal quality, which is relevant for developers, and external quality, which is relevant to end users.

The main emphasis lays on the affordability of the model in practice. To this end, a number of affordability requirements are defined and used as guidelines for constructing two software quality indexes: **IPQI**[®] (for internal quality) and **EPQI**[®] (for external quality). These indexes are based on common software metrics, easily available in practice via automatic extraction means. The indexes are designed to offer a convenient way of monitoring quality at different levels in the organization, and different phases in the product life-cycle. They have enough resolution to monitor a wide range of quality affecting actions, and facilitate traceability of quality issues to the root cause.

Case studies are currently performed with industrial partners to validate the model in a real-world environment. The intermediate results are very promising, yet they show that feasibility and practicality are not the same. While most organizations have testing and metric extraction frameworks in place, the results are not centralized and consistently stored. Therefore, a relatively significant effort is required to set-up the validation infrastructure in a given environment. This requires management commitment which can only be obtained in companies where product quality is already considered to be of strategic interest. It is, however, very important to apply and study the model also on products developed in contexts where quality is less relevant. This would enable a better tuning of the factors that require empirical calibration (e.g., the “weakest link” bias profiles) and would facilitate establishing of a set industry-wide quality standard levels.

Authors

Dr. Lucian Voinea (voinea@software-benchmarking.org) is managing director of SolidSource BV (www.solidsourceit.com) and specialized in code analysis and visualization.

Dr. Hans Sassenburg (sassenburg@sw-benchmarking.org) is managing director of SE-CURE AG (www.se-cure.ch) and specialized in software measurement and analysis (including benchmarking).

References

R. Bache, G. Bazzana, “Software metrics for product assessment”, McGraw Hill, London, 1993.

V. Basili, B. Boehm, "COTS-Based Systems Top 10 List." IEEE Software 34, no. 5, May 2001, pp. 91-93.

B. Beizer, "Software testing techniques", Van Nostrand Reinhold, New York, 1990.

N.E. Fenton, M. Neil, "A Critique of Software Defect Prediction Research", IEEE Transactions on Software Engineering, Vol. 25 (5), 1999.

S.S. Gokhale, et al., "Important Milestones in Software Reliability Modeling", Communications in Reliability, Maintainability and Serviceability, SAE International, 1996.

M.H. Harrold, "Testing Evolving Software", Journal of Systems and Software, special issue of top scholars in the field of Systems & Software Engineering (1993-1997), Vol. 47 (2-3), pp.173-181, 1999.

LaQuSo, "Software Product Certification: Process Behavior from Source Code Certificate", 2008.

N. G. Leveson, "Safeware, System Safety and Computers", Reading, MA: Addison Wesley, 1995.

T. J. McCabe, "A Complexity Measure", IEEE Transactions on Software Engineering, Vol. 2, pp. 308-320, 1976.

J.D. Musa, et al., "Engineering and Managing Software with Reliability Measures," McGraw-Hill, 1987.

J.A. Whittaker, "What Is Software Testing? And Why Is It So Hard?" IEEE Software, January/February, 2000.

Nicholas G. Rupp , "The Attributes of a Costly Recall: Evidence from the Automotive Industry," Review of Industrial Organization, Vol. 25, 2004.

ISO, "ISO/IEC 9126-1:2001(E): Software engineering — Product quality —Part 1: Quality model," International Organization for Standardization, 2001

ISO, "ISO/IEC TR 9126-2: Software engineering — product quality — part 2: External metrics," International Organization for Standardization, 2003.

ISO, "ISO/IEC TR 9126-3: Software engineering — product quality — part 3: Internal metrics," International Organization for Standardization, 2003.

D. M. Coleman, *et al.*, "Using metrics to evaluate software system maintainability," IEEE Computer, vol. 27, 1994.

H. Zuse, "A Framework of Software Measurement." Hawthorne, Walter de Gruyter & Co., 1997.

N. E. Fenton and M. Neil, "Software metrics: roadmap." in ICSE - Future of SE Track, 2000

W. Suryn, *et al.*, "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.

A. Abran *et al.*, ISO-based Model to Measure Software Product Quality. Institute of Chartered Financial Analysts of India (ICFAI), ICFAI Books, 2007

R. Geoff Dromey, "A Model for Software Product Quality," *IEEE Trans. Software Eng.* 21(2), 1995

S. McConnell, *Code Complete*, Microsoft Press, 1993

I. Heitlager, *et al.*, "A Practical Model for Measuring Maintainability," QUATIC '07: Proceedings of the 6th International Conference on Quality of Information and Communications Technology, IEEE Computer Society, 2007

Peter G. Bishop, "Estimating Residual Faults from Code Coverage," *Computer Safety, Reliability and Security*, 2434/2002, Springer, 2002

X. Cai, and M.R. Lyu, "The effect of code coverage on fault detection under different testing profiles," Proceedings of the 1st international workshop on Advances in model-based testing, ACM Press, 2005

Y.K. Malaiya, and J. Denton, "Estimating Defect Density Using Test Coverage", Technical Report CS-98-104, Colorado State University Press, 1998

W. Stevens, G. Myers and L. Constantine, "Structured Design", *IBM Systems Journal*, 13 (2), 115-139, 1974.

M. Hitz and B. Montazeri, "Measuring Coupling and Cohesion In Object-Oriented Systems," *Proc. Int. Symposium on Applied Corporate Computing*, 1995

S.R. Chidamber and C.F. Kemerer, "A Metrics suite for Object Oriented design," M.I.T. Sloan School of Management E53-315, 1993

B. Henderson-Sellers, L. Constantine and I. Graham , “Coupling and Cohesion (Towards a Valid Metrics Suite for Object-Oriented Analysis and Design)”, *Object-Oriented Systems*, 3(3), 1996.

J.M. Bieman and & K. Byung-Kyoo: “Cohesion and reuse in an object-oriented system,” *Proceedings of the 1995 Symposium on Software*, ACM Press, 1995

RTCA/DO-178B, "*Software Considerations in Airborne Systems and Equipment Certification*", 1992.