

# SBO

SOFTWARE BENCHMARKING

## epqi<sup>®</sup>

“Making Software Quality Transparent”

Author(s): Lucian Voinea  
Hans Sassenburg  
Version: 1.0  
Latest update: 12-Oct-2009

## Contents

1	Introduction .....	3
1.1	General.....	3
1.2	What is Quality?.....	4
1.3	Quality Model .....	4
2	Software Reliability .....	5
2.1	General.....	5
2.2	Test Coverage.....	5
2.3	Conclusion.....	7
3	Software Maintainability .....	8
3.1	General.....	8
3.2	Test Coverage.....	9
3.3	Conclusion.....	9
4	Aspects Considered.....	10
4.1	General.....	10
4.2	Testing Levels .....	10
4.3	Binary Units.....	11
5	What Makes a Good Index/Metric? .....	13
5.1	General.....	13
5.2	Criteria .....	13
6	Unit Testing Quality .....	15
6.1	General.....	15
6.2	Formula.....	15
6.3	Practicality.....	16
7	Integration Testing Quality .....	18
7.1	General.....	18
7.2	Formula.....	18
7.3	Practicality.....	19
8	Binary Unit Quality.....	21
8.1	General.....	21
8.2	Formula.....	21
8.3	Practicality.....	22
9	epqi® .....	23
9.1	General.....	23
9.2	Formula.....	23
9.3	Index Format .....	23
9.4	Practicality.....	24
10	Example.....	26
11	Conclusions .....	27
11.1	Model.....	27
11.2	Case Studies .....	27
11.3	Open issues .....	27
	Appendix: References .....	29

# 1 Introduction

## 1.1 General

In today's dynamic software development climate, with its ever-increasing pressure on efficiency and economic payoffs, there needs to be an increased focus on the quality of the software products that are being developed. Software quality is an essential concern in software industry. Software is now used in many demanding applications and software defects have caused serious damage and even physical harm. While defects in financial or word 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 be dangerous. People have been killed by defective software (Leveson [1995]).

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 like CMMI®. Augmenting the evaluation of an organization's process with an analysis of the quality of the software products that are produced by that process can identify areas where process improvement would result in improved product quality. However, process improvement should not just focus on plotting an organization's progress against a model of process maturity: better product quality (effectiveness) and increased productivity (efficiency) are the real objectives. The keys to better software are not simply 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 quality be made explicit during the development process to software developers and testers?
- How can a software manufacturer know whether a software product is ready for releasing?
- How can potential buyers be convinced of the quality of a software product?

Because software quality is so important, we need to first discuss what we mean by **software quality**.

---

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

## 1.2 What is Quality?

Software quality is frequently interpreted as “absence of defects”. This view of software quality focuses mainly on **reliability** as the primary characteristic and can be defined as “the probability of failure-free operation of a computer program in a specified environment for a specified time” (Musa et al. [1987]). 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. In addition, the increased focus on reuse as a means to reduce cost and time to field increases the value of software components that can be easily integrated in new software products. So, the second important characteristic of product quality we will consider here is **maintainability**.

In summary, the two quality characteristics which command our attention are reliability and maintainability. High quality in our definition means that:

- The likelihood to fail is low (demonstrates high reliability).
- The likelihood to fix failures and change/add functionality against low cost is high (demonstrates good maintainability).

## 1.3 Quality Model

In this document, a quality model is proposed to express the quality of a software product into one single normative figure: the **external product quality index** or **epqi®** for short [epki:].

This index should encode the quality of a software product, hereby supporting:

- Software developers and testers to deliver high-quality software by assessing and improving quality during product development and maintenance.
- Software manufacturers to certify the quality of a software product prior to releasing it to its intended customer(s) and/or end-user(s).
- Potential buyers to assess the quality of end-user software products or third-party software stacks.

The proposed model strictly focuses on the quality of the product: source code and binary units, for which no source code is available. Related artifacts like documentation, as proposed for instance by LaQuSo [2008], are not taken into account. Such artifacts are considered to be an indirect approach to achieve software product quality as well and should therefore not be included in the assessment and certification of the quality of a software product.

## 2 Software Reliability

### 2.1 General

Software reliability, defined as the probability that a software product will operate without failure under given conditions for a given time interval, is an important non-functional requirement to take into account when the release trade-off question is raised. If testing, as the last project stage, is stopped too early, significant defects could be released to intended users and the software manufacturer could incur the post-release cost of fixing resultant failures later. If testing proceeds too long, the cost of testing and the opportunity cost could be substantial.

So, budget and time (release now without spending more, but with a certain number of defects) are traded against improved quality through continued testing (release later with less defects but additional test budget). In theory, software reliability estimation models offer the possibility of determining the optimal release time, and also the possibility of determining the effects of a time, or budget, constraint. Where a software manufacturer is faced with a time-constraint (fixed schedule delivery) or a budget-constraint (limited cost), it is possible to make predictions of the operational cost based on the reliability level at release time. A software manufacturer might further be faced with a reliability-constraint (minimal reliability requirement), and these models can help determine the time and test cost for reaching this requirement, and the expected operational cost after product release.

### 2.2 Test Coverage

Software reliability estimation models (also referred to as reliability growth models) evaluate current and future reliability from faults, beginning with the integration, or system testing, of the software. The estimation is based on test data. These models attempt to statistically correlate defect detection data with known functions, such as an exponential function. The usefulness of software reliability estimation models has been heavily criticized. Criticism is twofold:

- Software reliability estimation models evaluate current and future reliability from faults, beginning with the integration, or system testing, of the software. The estimation is based on test data. These models attempt to statistically correlate defect detection data with known functions, such as an exponential function. The usefulness of the software reliability estimation models has been heavily criticized. Most models assume a way of working that does not reflect reality (Whittaker [2000]), meaning that the quality of assumptions is low. As a result, several models

- can produce dramatically different results for the same data set (Gokhale *et al.* [1996]).
- Fenton and Neil [1999] studied the most-widely used models and observe many shortcomings. They conclude that, as a result, these models provide little support for determining the reliability of a software product. Their study also shows that the number of pre-release faults is not a reliable indicator of the number of post-release failures.<sup>2</sup> The problem is that many software manufacturers use the pre-release fault count as a measure for the number of post-release failures, e.g. the reliability of the released product

As an alternative, code coverage, as an indicator to show how thoroughly software has been stressed, has been proposed and is widely employed to represent fault coverage. A comparison among various studies on the impact of code coverage on software reliability is shown in Table 1.

Reference	Findings
Horgan [1994] Frankl [1988] Rapps [1988]	High code coverage brings high software reliability and low fault rate.
Chen [1992]	A correlation between code coverage and software reliability was observed.
Wong [1994]	The correlation between test effectiveness and block coverage is higher than that between test effectiveness and the size of test set.
Frate [1995]	An increase in reliability comes with an increase in at least one code coverage measure, and a decrease in reliability is accompanied by a decrease in at least one code coverage measure.
Cai [2005]	Code coverage contributes to a noticeable amount of fault coverage.

Table 1: Relation of code coverage to fault coverage.

Despite the observations of a correlation between code coverage and fault coverage, a question is raised: can this phenomenon be attributed to a causal dependency between code coverage and fault detection, or is it just coincidental due to the cumulative nature of both measures? An experiment involving Monte Carlo simulation was conducted on

<sup>2</sup> Beizer [1990] uses an analogy with pesticide to illustrate the difficulty in software testing, known as the Pesticide Paradox: *Every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffectual.* By eliminating the (previous) easy bugs one allows an escalation of features and complexity, but this time one is faced by super, subtler, bugs.

the assumption that there is no causal dependency between code coverage and fault detection (Briand [2000]). The testing result for published data did not support a causal dependency between code coverage and defect coverage. Nevertheless, many researchers consider coverage as a faithful indicator of the effectiveness of software testing results.

### **2.3 Conclusion**

We conclude that there is a strong correlation between code (test) coverage and reliability (fault) coverage.

## 3 Software Maintainability

### 3.1 General

Software products with a long-term lifespan need to be maintained, which is a term used to describe both the removal of undesired functionality (defects) and the addition of desired functionality (done typically to catch up with or pass the competition), and Without maintenance, customers/end-users abandon a product because eventually, it fails to meet their needs as well as the competition, or they simply tire of putting up with defects in the application.

Software maintenance is usually the costliest part of software development, because it is an ongoing expense and because the cost generally increases with time. This implies that to minimize software development costs, one strategy is to write software that is easy to maintain.

The *IEEE 1219* standard provides a seven-step model for the process of good quality software maintenance, where the requirements for control, management, planning and documentation are highlighted (IEEE [1998]), and the appendix provides methods and tools, and discusses reverse engineering. The standard distinguishes the following categories of maintenance:

- Corrective maintenance: reactive modification of a software product; performed after delivery to correct discovered faults.
- Adaptive maintenance: modification of a software product; performed after delivery to keep the computer program usable in a changed, or changing, environment.
- Perfective maintenance: modification of a software product; performed after delivery to improve performance or maintainability. The improvement of performance can be described as evolutive maintenance and the improvement of maintainability as preventive maintenance.

Addressing reliability as the only or main quality characteristic is dangerous. Considering the total life-cycle cost of a software product, focus should not only be on the short-term operational cost for repairing failures due to poor reliability (corrective maintenance cost), but also on the expected future cost for extending the product with additional functionality (adaptive and perfective maintenance cost).

### **3.2 Test Coverage**

Cyclomatic complexity is a static software metric pioneered by McCabe [1976]. The cyclomatic complexity number is a measure of how many paths there are through a unit of code. It serves as a rough measure of code complexity and as a count of the minimum number of test cases that are required to achieve full code coverage of the unit. The number of execution paths through a unit is directly related to the maintainability of the unit.<sup>3</sup>

When the determination has been made that a unit is complex, there are two ways available to mitigate the associated risk:

- Increase the number of unit tests.
- Refactor the offending code into smaller pieces to reduce its complexity; this spreads the complexity over smaller, more manageable and therefore more testable and maintainable units.

### **3.3 Conclusion**

We conclude that there is a (strong) correlation between code (test) coverage and maintainability. High test coverage can normally only be obtained by reducing cyclomatic complexity, which contributes to better maintainability.

---

<sup>3</sup> There is no convincing evidence to show that complexity metrics are good predictors of reliability. However, there is a growing body of evidence that some of these metrics may be useful in outlier analysis especially when grouped together (Bache and Bazzana [1993]).

## 4 Aspects Considered

### 4.1 General

In the previous two Chapters, we concluded that reaching high test coverage contributes to both higher reliability and better maintainability. These are considered the two main characteristics of software quality.

We now have to define at which levels we would like to reach high test coverage.

### 4.2 Testing 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 (see Section 2.2).
- 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 (Whittaker [2000]). 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.<sup>4</sup>

- Another complicating factor stems from the dynamic nature of software (Harrold [1999]). 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 (Whittaker [2000]). To account for this possibility, testing should be restarted. However, the incurred costs of re-executing all previous test cases are often prohibitive.

We conclude that a requirement for obtaining high product quality in terms of reliability and maintainability, consideration of quality should be integrated from the very beginning of implementation and testing phase: unit and integration level. 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. Put differently, 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. We therefore restrict the scope of the model to unit and integration testing and do not take system testing into account. It enables one to define start criteria for system verification and validation ensuring that the underlying units have sufficient quality and can be integrated.

### 4.3 Binary Units

As software development has grown more complex, it has become a multi-tier process with many parties involved. Software products are now comprised of code developed by the software manufacturer itself, open source, sub-contractors, offshore providers and third-party components. Lower up-front costs, and a belief that the cost savings extend

---

<sup>4</sup> 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.

throughout the software product's life cycle are primary motivators in the shift from fully custom-designed to integrated products.

The disadvantages associated with purchased third-party software (binary units) include the absence of source code. The burden of minimizing risk and controlling operational cost from low-quality binary units has been placed largely on their buyers (Basili and Boehm [2001]). In most cases organizations do not have any insight into what vulnerabilities exist in these binary units, resulting in an unacceptable level of unbounded risk. Testing such applications before 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 are restricted to analyzing a small sub-set of the purchased software which results in unbounded risks to the organization.

The quality of such binary units will influence the quality of the overall software product. Therefore, the model should take into account the quality contribution of each binary unit when computing the overall **epqi**®. This should preferably be done in a way without asking the vendor to expose any of their intellectual property in the form of source code.

## 5 What Makes a Good Index/Metric?

### 5.1 General

Software metrics are an integral part of the state-of the-art practice in software engineering. More and more customers are specifying software and/or quality metrics reporting as part of their contractual requirements. Industry standards like ISO 9000 and industry models like CMMI® include measurement. Companies are using metrics to better understand, track, control and predict software projects, processes and products.

The term *software metrics* means different things to different people. We define software metrics as: "The continuous application of measurement-based techniques to the software development process and its products to supply meaningful and timely management information, together with the use of those techniques to improve that process and its products" (Goodman [1993]). Software metrics can provide the information needed by engineers for technical decisions as well as information required by management.

### 5.2 Criteria

If a metric is to provide useful information, everyone involved in selecting, designing, implementing, collecting, and utilizing it must understand its definition and purpose. We define four straight-forward criteria that address especially the practicality of metrics:

- **Simple:** Most software systems are quite complex and the people who work on them are usually quite smart, so it seems both reasonable and workable to use complex metrics. Although complex metrics may be more accurate than simple ones, and most developers will be able to understand them (if they are willing to put the time into it), the popularity, effectiveness, and usefulness of most metrics (software or otherwise) is inversely proportional to their complexity.<sup>5</sup>
- **Positive:** A metric is considered to be positive if one wants the quantity it measures to go up.

---

<sup>5</sup> The Dow Jones index is a good example of this effect. It is a very old metric and is necessarily simple. There are thousands of stocks that can be tracked and the Dow Jones takes into account only a limited number of stocks. But because it is simple and it seems to track a portion of the stock market well enough, it is a widely reported, understood, and followed market index.

- **Controllable:** The success of an effort should be tied to one or more metrics that can be controlled. Each (combination of) metric(s) should enable one to define measures to positively influence the metric(s).
- **Automatable:** If calculating a metric requires manual effort it will quickly turn into a chore and it will not be tracked as frequently or as accurately as it should be. It is essential that whatever one decides to measure can be easily automated and will require little or no human effort to collect the data and calculate the result.

These criteria regarding practicality will be assessed when defining the **epqi**® and its underlying metrics.

## 6 Unit Testing Quality

### 6.1 General

Code coverage at unit level should also be brought as close to 100% as possible, before a unit is passed forward and contributed to the toolkit. The reason is that the relevance of a test passing is only significant at the level of the code coverage of the test. In other words, a test that passes but that only exercises 20% of the code in a unit cannot be claimed to be a sufficient demonstration of the implementation correctness. Such test will only prove that 20% of the unit works as advertised.

To compute the quality of the unit testing for a source code unit, we propose using a branch based code coverage measure that is biased with statement coverage (SC). This should ensure that both branches and statements are sufficiently covered in order to obtain high quality unit testing. We propose using as branch measure the MC/DC (modified condition decision coverage) measure (MCDC), as this is already part of the DO-178B Level A aviation standard (RTCA [1992]).

The model should give total freedom in choosing one's own definition for a unit. The smaller the amount of code that you call a unit, the higher the reported quality values should be. This means that if the quality of your system is not high enough, you will be encouraged to break it in smaller pieces when trying to improve the reported figures. However, by doing so you'll also increase the coverage of the tests, and increase the probability of finding bugs. Therefore the quality will also increase in a natural way. The opposite is also ok. If your quality is very good, you can choose the highest unit level you like, which will make your life easier when monitoring the epqi®.

### 6.2 Formula

The unit testing quality ( $QU_i$ ) associated with a source code unit ( $i$ ) will be given by the formula:

(1)

#### Observations:

This measure favors the decomposition of a system in small size units. The smaller the unit, the easier it becomes to obtain high code coverage during testing.

To compute the overall quality of the unit testing aspect in the system we should consider the way the quality of a system is perceived. Namely, the perceived quality of a system is greatly influenced by its ‘weakest link’. Consequently, a formula that aggregates the quality levels of individual system units should emphasize/favor low quality units.

To accommodate this aspect in the overall value for unit testing quality, we introduce a quality based weighting factor:  $wU_i$ . This factor should be used for aggregating the individual  $QU_i$  values of the units in one overall value, and has to be determined experimentally. Figure 1 gives overall guidelines for setting it.

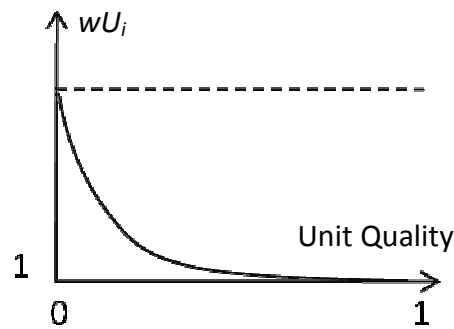


Figure 1: Unit testing quality - guidelines for the quality bias factor.

The overall unit testing quality ( $QU$ ) is given by the formula:

$$\text{_____} \tag{2}$$

**Observation:**

The  $QU$  value can be mapped on the A-E range in the final **epqi**® to give an indication of the overall unit testing quality in the end product (to be discussed in Section 9.3).

**6.3 Practicality**

Does the metric  $QU$  match the defined criteria? As can be seen in Table 2, all criteria receive a positive score.

Criterion	Score	Clarification
Simple	+	Branch and statement coverage are widely known principles in testing. Code coverage measurements match very well the organization-based perspective of code that software engineers have. At the same time, being a percentage based figure, it is easy to use from a management point of view. Consequently, code coverage measurements form a good vehicle for communicating facts across expertise boundaries in software projects.
Positive	+	The metric is positive as the objective should be to obtain a high(er) value.
Controllable	+	Both code coverage measures work by detecting the code that is not executed during testing. By reporting the exact location of not executed code, coverage measurements enable software engineers to make correlations between the set of execution scenario parameters of an application and the actual execution path through the program. Via new/extended test cases, the engineers can subsequently influence this execution path, such that a better coverage is obtained during testing.
Automatable	+	The two coverage measurements can be fully automated. Both for branch and statement coverage, tooling is available.

Table 2: Practicality scores for the *QU* metric.**Observations:**

- Using the MC/DC (modified condition decision coverage) measure for branch coverage may give the model a lift towards adoption. However, if this metric is not widely known and used, it may be replaced with normal branch coverage.
- The easier is to make correlations between execution scenario parameters of an application and the execution path through the corresponding program, the easier it gets to control/increase the coverage measurements, and consequently the *QU* measure. To reduce the amount of effort required to improve the *QU*, one should aim at bringing unit testing as close as possible to implementation, both in time and people resources. A Test Driven Development (TDD) approach, for example, would be a good alternative in this respect. Other development strategies are also supported, although, in some cases (e.g., pure waterfall) improving the *QU* values would be difficult to achieve at testing time.

## 7 Integration Testing Quality

### 7.1 General

To compute the quality of the integration testing of a source code unit we need to consider the possible interactions it has with the other units. A high quality integration testing should address all possible interactions. 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 statement coverage. However, even for a high coverage value, it is possible that the unit 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 statement coverage value. In both cases, large inconsistencies can appear between the measured statement coverage and the actual quality of the integration.

To focus the statement coverage approximation of relevant states for interaction, we propose combining the coverage measure 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 give places where variables depending on the called function parameters are used. The coverage analysis should be then applied only on the identified functions. For a better coverage of possible unit states, one should use the same coverage measure as for unit testing quality.

### 7.2 Formula

The integration testing value ( $QI_i$ ) for one unit can be then computed as a weighted sum of all relevant integrations  $j$  for a given unit. Similarly to Section 6.2, the weights ( $wIU_j$ ) should be given by the  $QI_j$  value of the associated piece of code on which code coverage is performed (i.e., identified by dataflow analysis) and should be adjusted experimentally.

(3)

where  $QU_j$  is similar to the  $QU_i$  measure from Section 6.2, but applied to the piece of code that is identified by the data flow analysis.

**Observation:**

Subsequently, the overall integration testing quality  $QI$  can be computed as the weighted average of the integration quality values specific to each unit. The weighting factor  $w_i$  of each unit should be quality based and determined experimentally, similarly to the one in Section 6.2.

(4)

**Observation:**

The  $QI$  value can be mapped on the A-E range in the final **epqi**® to give an indication of the overall unit integration testing quality in the end product (to be discussed in Section 9.3).

**7.3 Practicality**

Does the metric  $QI$  match the defined criteria? As can be seen in Table 3, all criteria receive a positive score.

Criterion	Score	Clarification
Simple	+	Uses code coverage and, hence, it is easy to accept by different roles in the project. See Section 6.3.
Positive	+	The metric is positive as the objective should be to obtain a high(er) value.
Controllable	+	By adjusting the test application scenarios, a tester can influence the execution path of a program, and hence its code coverage. See also Section 6.3.
Automatable	+	Code coverage measurements can be fully automated.

Table 3: Practicality scores for the  $QI$  metric.

### **Observation**

The main challenge of controlling code coverage based measures is to make correlations between the set of execution scenario parameters of an application and the actual execution path through the associated program. Consequently, the closer the attempt to improve  $QI$  is to the initial implementation (in time and people), the easier it is to perform it. This tends to favor smaller organizations where integration testing is performed by the same engineers that have initially implemented the unit.

To make the controllability requirement achievable in practice in larger organizations where (possibly) different teams are responsible for initial development and integration, one has to migrate the correlation effort from integration testing time to previous stages.

To this end, one could reuse the test cases/parameters settings from unit testing for implementing integration testing cases. With such an approach, the correlation effort will be spent on extracting the execution parameters from previous test cases that are relevant for increasing the  $QI$  value. Automated tool support can be used to restrict the set of candidate unit test cases to the code regions that are relevant to integration testing according to the data flow analysis.

## 8 Binary Unit Quality

### 8.1 General

It is not possible/practical to reliably assess the code quality of a binary unit using a black box testing approach. Therefore, the quality of such a unit should be assessed by its developers using a white box testing approach at development time, and passed further together with the binary unit. The quality value should be assessed by developers using the formula presented in the next section.

### 8.2 Formula

The overall binary unit quality ( $QB$ ) in a final software product can be estimated by using a formula similar to the one for unit testing quality (see Section 6.2). In place of  $QU_i$  one can use the value provided with the binary unit.

$$\text{_____} \tag{5}$$

#### Observation:

The weighting factor  $wB_i$  of each binary unit should be quality based and determined experimentally, similarly to the one in Section 6.2.

The overall value can be mapped then on the A-E interval in the final **epqi**® (to be discussed in Section 9.3).

Binary units do not have an integration factor in the overall *epqi* formula. The integration aspects related to binary units are considered to be included in:

- The integration tests performed by the developers of the binary units;
- The integration tests on the source code units of the current system.

### 8.3 Practicality

Does the metric *QB* match the defined criteria? As can be seen in Table 4, all criteria receive a positive score.

Criterion	Score	Clarification
Simple	+	The <i>QB</i> value is based on the <i>epqi</i> value which is intended to be easily accepted by the software engineering community at all levels.
Positive	+	The metric is positive as the objective should be to obtain a high(er) value.
Controllable	+/-	Development models that use binary object are most often based on the assumption that the implementation of the binary unit is not subject to change. From this point of view, the only mechanism that is available within the project for increasing the quality level of a binary unit is to replace this with a better quality one, provided that the new unit offers the same functionality. In the very seldom case that a project can influence the implementation of a binary unit, the controllability aspect is guaranteed by a combination of the controlling activities presented in Sections 6.3 and 7.3.
Automatable	+	Once the individual <i>epqi</i> values are available (true by definition) the <i>QB</i> value can be automatically computed.

Table 4: Practicality scores for the *QB* metric.

## 9 epqi®

### 9.1 General

The total **epqi**® has to take into account the unit testing, integration testing and binary unit quality aspects described in the previous Sections.

To have a practical use, such a formula should support composition of higher level *EPQI* values from lower level ones. This would be necessary, for example, to integrate the quality of the binary units in the final figure. Composition of *epqi* values would also enable more convenient and practical ways of managing and monitoring the quality of a software product.

### 9.2 Formula

From this point of view, the proposed calculation formula for the overall *epqi* value is:

$$\text{epqi} = \frac{\sum (wO_i \cdot epqi_i)}{\sum wO_i} \tag{6}$$

where

$$wO_i = \frac{1}{epqi_i} \tag{7}$$

and  $wO_i$  is a weighting factor based on  $epqi_i$ ; that has to be set experimentally, similar to the weighting factors presented in the previous sections.

**Observation:** *epqi*

### 9.3 Index Format

The **epqi**® is a single normative figure (between 0 and 1000) to characterize the overall 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 ingredients mentioned above. This would be interesting to know for understanding 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 could consider 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 consider the purchase. If the quality of the source code level is however low, he will probably not.

Therefore, we also propose presenting the **epqi**® together with a three letter pattern that presents the individual contribution of each of the three quality aspects considered. By requiring that all three individual aspects are expressed on the same scale between 0 and 1000, we can define a range [A (Excellent) – E (Very poor)] for both the **epqi**® and the quality of each aspect. See Table 5.

Letter	Interval	Certification level	Typical Industry Segment <sup>6</sup>
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 5: Mapping between letters, quality interval and rating.

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).

### 9.4 Practicality

Does the metric *epqi* match the defined criteria? As can be seen in Table 6, all criteria receive a positive score.

<sup>6</sup> These are assumptions at this stage. Further research towards the requirements as well as industry averages for each industry segment is needed.

Criterion	Score	Clarification
Simple	+	It is based on the basic quality measurements $QU$ and $QI$ for unit testing and integration testing defined in the previous sections.
Positive	+	The metric is positive as the objective should be to obtain a high(er) value.
Controllable	+	By controlling the $QU_i$ and $QI_i$ values corresponding to the integrated units, one can control the overall $epqi$ .
Automatable	+	All entities involved in computing $epqi$ can be automatically computed; hence, the aggregation can be also automated.

Table 6: Practicality scores for the  $epqi$  metric.

## 10 Example

Consider a software product, consisting of two units and two binary units, with the characteristics as in Table 7:

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 7: Software product characteristics.

We can compute the following indexes:

$$QU = (10/18 * 0.1 + 8/18 * 0.2) * 1000 = 144.44 \quad (E)$$

$$QI = (2 * 0.6/3 + 0.9/3) * 1000 = 700 \quad (B)$$

$$QB = 1.5/2.5 * 700 + 1/2.5 * 800 = 740 \quad (B)$$

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, see Figure 2.

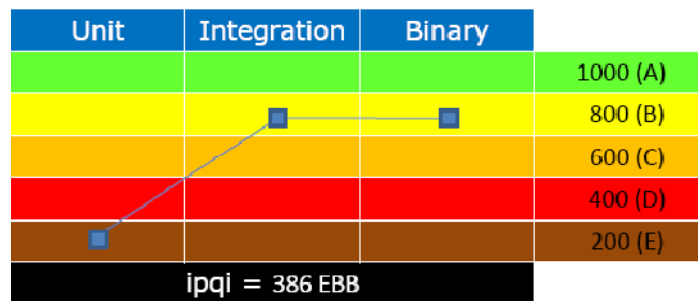


Figure 2: Calculated **epqi**®.

## 11 Conclusions

### 11.1 Model

The proposed model defines the **epqi**® and its underlying metrics. The objective of this index is to support assessment and certification exercises in the area of software product quality. The model supports:

- Software developers and testers to deliver high-quality software by assessing and improving quality during product development and maintenance.
- Software manufacturers to certify the quality of a software product prior to releasing it to its intended customer(s) and/or end-user(s).
- Potential buyers to assess the quality of end-user software products or third-party party software stacks.

As far as we can see now, both the index and its underlying match the defined criteria for practicality: simple, positive, controllable and automatable.

### 11.2 Case Studies

Case studies should demonstrate that the model is feasible and practical in a real-world environment. Feasibility and practicality are not the same as the requirement that every software manufacturer should be able to measure the **epqi**® of his portfolio with limited effort. It is expected that very few software manufacturers use code coverage measures during software development and maintenance. In addition, the development environment used will not always include the required tooling to make automated measurements of coverage possible. Knowing this, it might well be true that case studies will be initially limited to those manufacturer organizations where appropriate tooling and coverage expertise is available. It might even be considered to restrict the validation the model to smaller parts of a software product with a strong focus on unit testing quality only.

### 11.3 Open issues

Known open issues at this stage are:

- **Model and formulas.** The model and the used formulas are chosen in such a way that the quality of an entire software product as well the quality of the different

- aspects (unit testing, integration testing, binary units) can be expressed on an identical scale. In addition, it has been strived to propose a model that matches intuition. It might however be considered to have the entire model checked by one or more mathematical experts.
- **Bias factors.** The bias factors defined for unit testing quality need to be set experimentally. Initial values need to be investigated prior to releasing the model, so people can start using it.
  - **Technology.** The coverage concepts used are quite generic and the model is assumed to be applicable to (the combination of) any language(s). This has to be verified.
  - **Unit definition.** The formulas presented in this document favor the decomposition of a system in small units that are easy to test. However, the definition of a unit can be very different from one organization to another. The implications of choosing the scope of a unit on the final *epqi* value needs to be investigated in more detail. One issue that has already been identified is related to integration testing. When small units are combined in larger ones, their integration may not be taken into account (for example, stubs could be used during unit testing of the combined unit to avoid testing the interaction aspects of the two original units). To avoid this, one should consider in the integration testing phase not only external integration aspects but also internal ones. That is, if a component uses internally interfaces it declares public, it should test their internal usage too. Practically, this means identifying the places where a public interface is used internally and treating those places as an instance of interaction across unit borders.

## Appendix: 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.

L. Briand and D. Pfahl, "Using Simulation for Assessing the Real Impact of Test Coverage on Defect Coverage," *IEEE Transactions on Reliability*, vol. 49, no. 1, March 2000, pp. 60-70.

X. Cai and M.R. Lyu, "The Effect of Code Coverage on Fault Detection Under Different Testing Profiles," *ICSE 2005 Workshop on Advances in Model-Based Software Testing (A-MOST)*, St. Louis, Missouri, May 2005.

M. Chen, M.R. Lyu, and E. Wong, "Effect of Code Coverage on Software Reliability Measurement," *IEEE Transactions on Reliability*, vol. 50, no. 2, June 2001, pp.165-170.

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

P.G. Frankl and E.J. Weyuker, "An Applicable Family of Data Flow Testing Criteria," *IEEE Transactions on Software Engineering*, vol. 14, no. 10, October 1988, pp. 1483-1498.

D. Frate, P. Garg, A. Mathur, A. Pasquini, "On the Correlation between Code Coverage and Software Reliability", *Sixth International Symposium on Software Reliability Engineering*, 1995.

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

P. Goodman, "Practical Implementation of Software Metrics", McGraw Hill, London, 1993.

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.

J.R. Horgan, S. London, and M.R. Lyu, "Achieving Software Quality with Testing Coverage Measures," *IEEE Computer*, vol. 27, no.9, September 1994, pp. 60-69.

IEEE, "IEEE Standard for Software Maintenance", IEEE Std. 1219, The Institute of Electrical and Electronics Engineers, 1998.

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.

S. Rapps and E.J. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Transactions on Software Engineering*, vol. 11, no. 4, April 1985, pp. 367-375.

RTCA/DO-178B, "*Software Considerations in Airborne Systems and Equipment Certification*", December 1, 1992.

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

W. E. Wong, J. R. Horgan, S. London and A. P. Mathur, "*Effect of test size and block coverage on fault detection effectiveness*," Fifth International Symposium on Software Reliability Engineering, 1994, pp. 230-238.