Quality in software development: A pragmatic approach using metrics

Daniel Acton[*], Derrick G Kourie[†], Bruce W Watson[†]

[*]Espresso Research Group, Department of Computer Science, University of Pretoria, South Africa
[†]Espresso Research Group, Department of Information Science, University of Stellenbosch, South Africa

ABSTRACT

As long as software has been produced, there have been efforts to strive for quality in software products. In order to understand quality in software products, researchers have built models of software quality that rely on metrics in an attempt to provide a quantitative view of software quality. The aim of these models is to provide software producers with the capability to define and evaluate metrics related to quality and use these metrics to improve the quality of the software they produce over time. The main disadvantage of these models is that they require effort and resources to define and evaluate metrics from software projects.

This article briefly describes some prominent models of software quality in the literature and continues to describe a new approach to gaining insight into quality in software development projects. A case study based on this new approach is described and results from the case study are discussed.

KEYWORDS: Software Development, quality, metrics, measurement

CATEGORIES: D., D.2, D.2.1, D.2.4, D.2.7, D.2.8, D.2.9

1 INTRODUCTION

Quality has many definitions in the literature. The Institute for Electrical and Electronic Engineers (IEEE) Standard Glossary of Software Engineering Terminology defines quality as "the degree to which a system, component, or process meets specified requirements or customer or user needs or expectations" [1]. The ISO9000 standard defines quality as "the degree to which a set of inherent distinguishing features fulfils a need or expectation that is stated, generally implied or obligatory" [2]. In the context of using metrics to measure software quality, the IEEE standard for a Software Quality Metrics Methodology defines quality as "the degree to which software possesses a desired combination of quality attributes" [3].

As discussed by Garvin [4], determining the quality of a product can be approached in a number of ways. The *transcendent approach* states that quality cannot be defined precisely, but is rather experiential, and one can learn to recognise quality through experience. The *product-based approach* holds that quality is measurable and related to the degree to which the product possesses some attribute. The *user-based approach* holds that quality is subjective, and is determined by the user of the product based on their perspective. The *manufacturing-based approach* equates quality to conformance to requirements: if a product is built as specified, it exhibits quality. The *value-based approach* holds that a quality product is a product that provides "performance at an acceptable price or conformance at an acceptable cost".

There are many approaches in the literature that attempt to gain insight into the quality of software. Industry standards, such as the ISO9000 [2] family of standards, CMMI [5] and the Personal Software Process (PSP) [6], can be seen as a distillation of industry experiences into "best practice". These industry standards stress the importance of process in the pursuit of product quality. The ISO25000 [7] family of standards, along with work by Boehm et al. [8], Hyatt and Rosenberg [9] and McCall et al. [10] propose measuring software quality using metrics. Other approaches, such as Cleanroom [11] address quality by ensuring that formal methods are followed in the construction of software. Approaches such as those proposed by Barkmann et al. [12] and Gousios and Spinellis [13] suggest the use of Software Configuration Management (SCM) systems to gain insight into software quality.

This article describes an approach to software quality measurement that relies on the processes often present in the course of a software project (such as requirements gathering, testing and development) and the software often used (such as an SCM system). It also describes a case study that observes the use of the proposed approach in a real-world software project.

Section 2 gives an overview of some of the ap-

Email: Daniel Acton `mja@danielacton.com`, Derrick G Kourie `dkourie@fastar.org`, Bruce W Watson `bruce@fastar.org`

proaches in the literature, along with analysis of these approaches. Section 3 describes a new approach to software quality measurement. Section 4 describes an implementation of the approach discussed in Section 3 and Section 5 discusses a case study that uses this new approach. Section 5 provides a conclusion and suggests areas of future work.

2 SOFTWARE QUALITY MODELS IN THE LITERATURE

This section briefly describes eight models in the literature that use metrics as a method of gaining insight into quality in software development. These models were chosen as they represent a diverse range of models, across time (ranging from 1977 until 2009), focus (metrics and process) and source (academic and industry). The section closes with a critique of these methods, indicating that there is a need for a more pragmatic approach to quality measurement which does not impose a heavy administrative burden on software developers and managers.

2.1 McCall's Quality Factors

In their 1977 technical reports to the United States Air Force [10], McCall, et al. describe an approach to assessing software quality using metrics. The approach relies on a set of factors and criteria which affect the quality of the software. Metrics are used to measure these criteria and give a quantitative indication of the quality of the software product across the development life cycle. An example metric measures the cross referencing that relates modules to requirements. This metric therefore attempts to measure the so-called *traceability* criterion, which in turn is believed to contribute to what the authors call the *correctness* factor.

2.2 Hyatt and Rosenburg's Software Quality Model

Hyatt and Rosenburg's model [9] relies on two core requirements: that the software works well enough and that it is available when needed. This model is based on the perspective of a project manager. It identifies areas of risk to the fulfilment of these two core requirements. Metrics are used to provide an indication of the risk in each of these risk areas. An example metric is the *number of weak phrases*, which attempts to measure the ambiguity of the requirements. This contributes to the so-called *ambiguity* attribute, which in turn affects the *requirements quality* goal. This goal is related to the *correctness risk* area: the higher the requirements quality, the lower the correctness risk.

2.3 Boehm's Characteristics of Software Quality

In their book "Characteristics of Software Quality" [8], Boehm et al. discuss an approach to analysing software quality based on a set of characteristics that are generally desirable for software to exhibit. These characteristics are grouped into three main areas: as-is utility (how well the software is usable as it is), maintainability (how easy it is to maintain the software), and portability (how easy it us to use the software in different environments). These three characteristics are used in order to gain some insight into the general utility of the software. In other words: software that is easy to use and maintain in different environments would be classified (according to Boehm, et al.) as having high utility to the user or purchaser of the software. These high-level characteristics are further classified into lower-level characteristics. Metrics are used to measure these characteristics. The metrics used in this model were chosen based on their correlation with software quality, the importance of having a high measure for that metric, the cost of automatically evaluating the metric, and the feasibility and completeness of automated evaluation. An example of a metric described by Boehm et al. is "Metric Number 1.12" which measures whether the code is readable or not. This metric contributes to the low-level characteristic of *legibility* which contributes to the intermediate characteristic of *understandability*, which contributes to the high level characteristic of *maintainability*.

2.4 Cleanroom Software Engineering

The origin of the *cleanroom* method is a paper titled "Certifying the Reliability of Software", in which Currit et al. [14] discuss a method of certifying the reliability of software before the software is released to users. The reliability of the software is measured by its Mean Time To Failure (MTTF). Currit et al. suggest that users of the software will not be as concerned with the number of defects in the software as they will be with the reliability of the software. Currit et al. assume that the output of the development phase of production is of high quality and focus more attention on testing the reliability of the software. This they do by performing a subset of operations that real users would perform in an environment that closely mimics that of the user.

Building on this initial work on software reliability, Mills et al. [11] turned their attention to the assumption that Currit et al. had made, namely that the development output exhibits high quality. Mills et al. proposed a more formal approach to the development effort in order to ensure that the output does indeed exhibit high quality. The result of this work was the *cleanroom* software engineering approach.

Cleanroom uses mathematical foundations in the design, specification and correctness verification of the product. The focus on a formal proof that the software works as required mollifies the need to debug developed outputs. The use of formal methods of object-based box-structure specification and design ensures that the specification and design of the product are well-defined. Function-theoretical methods are used to verify the correctness of software. Statistical usage testing is used to certify the quality of the product.

2.5 SCM-based approaches

Barkmann et al. [12] and Gousios and Spinellis [13] propose the automated evaluation of metrics based on a project's source code (stored in an SCM system) in order to gain insight into the quality of the software. Barkmann et al. propose an approach that uses external open source software code repositories to determine which metrics apply to a given project. Gousios and Spinellis propose an approach that allows software producers to use metrics based on more than just the source code (i.e. they could also include metrics based on information from bug-reporting software and email messages).

2.6 ISO25000

The ISO25000 family of standards (also known as Software product Quality Requirements and Evaluation or SQuaRE) [7] is an evolution of an earlier standard, ISO9126 [15] [16]. Both of these standards propose an approach to measuring software quality using metrics. The ISO25000 family describes a framework for quality models that categorises product quality into a hierarchical view: characteristics, subcharacteristics, sub-subcharacteristics, quality properties and quality measures. The family of standards also provides two instances of the model, namely a quality-in-use model (used to measure the quality of a product as it might be perceived by users of the product), and a product quality model (used to measure the quality of the finished product or interim products during development).

2.7 ISO9000

The ISO9000 [2] family of standards suggests that an organisation wishing to provide quality products or services should focus on the processes used to produce their product or provide their service. If the organisation implements, maintains and continually improves these processes, it is suggested that the outputs of these processes should be of consistent or improving quality. ISO9000 also promotes a focus on the requirements and satisfaction of the customer, and if the organisation aligns their processes with this focus, they should be producing software that customers perceive as satisfying their requirements and hence exhibiting consistent or increasing quality.

2.8 CMMI

CMMI focusses on the implementation and improvement of the processes involved in software development, and bases its focus on the *process management principle*: "the quality of a system is highly influenced by the quality of the process used to acquire, develop, and maintain it." [5]. An organisation that chooses to implement CMMI is effectively implementing a model that aims to make their processes more effective. The organisation progresses through CMMI levels from *initial* (where there are no—or very few—processes in place) through *managed*, *defined*, *quantitatively managed* and finally *optimizing*, where there are well-defined processes, and data from the processes are used to optimize the processes further. The suggestion CMMI makes is that if an organisation continually ensures that the processes meet business requirements and are improved over time, the quality of the output should be of a consistent and increasing level.

2.9 The Personal Software Process (PSP)

The PSP focuses on having the engineeers or developers producing the product be an integral part of driving the quality of the product. A project is made up of tasks and phases. In each phase, the engineers or developers must estimate the size and effort of the tasks in the phase. As they perform the work, they record information about their work, including any defects they may find or fix, and the actual time taken to complete the task. Once the work is complete, the estimations are compared with the actual data and as a result, future estimations should be more accurate. The metrics that are collected and calculated over the course of the project include quality measures. The indicators of quality in PSP include those based on defects, for example defects per thousand lines of code, defects detected per phase and defects fixed per phase, and those based on productivity. The PSP maintains that in order to strive for quality, engineers or developers must spend time and effort on detecting and fixing defects, particularly aiming to remove defects early and prevent defects through keeping records of defects, ensuring a good design, and reducing development time.

2.10 Analysis

The models discussed in this section strive in various ways to provide software producers with the ability to gain some insight into and/or control over the quality of their software product. Having a model that uses metrics and provides feedback about quality at different points in the development lifecycle is important to software producers so that they may gain some insight into the quality of their product and how they can maintain and increase the level of quality over time. The main disadvantages that the models described in this section have in common is that, for the most part, effort is required to define the metrics to be measured, measure them, and analyse their results for an insight into quality.

The approach proposed by McCall et al. has been criticised for being too subjective [17]. In addition, the assessment of quality is performed using a set of worksheets that require values for each metric. The effort required to implement this approach is a limiting factor to its implementation, and may yield results too late to be acted upon.

The approach proposed by Boehm et al. suggest that metrics be defined for a project based on factors like the intended users, characteristics and environments. This also creates a barrier to adoption since the effort to define them is required for every project.

While this approach does suggest automated evaluation of metrics, there could still be metrics for which manual evaluation is required.

The approach proposed by Hyatt and Rosenburg is biased to the view of the project manager, and this may mean that the approach does not cater for other perspectives.

It is important to bear in mind that industry standards are not the definitive instruction on a matter—they are merely guidelines based on experience. Thus, it is natural to expect that there are supporters and detractors of these standards. The efficacy of ISO9000 in increasing quality has been questioned by Stelzer et al. [18]. When organisations investigate implementing CMMI, there is evidence [19] [20] indicating that they may decide against implementing CMMI because the cost of implementing is seen as unjustified against the benefits. The effort requireds to implement the PSP approach could be a barrier to its adoption: engineers or developers need to estimate and calculate a number of metrics during their work, and while the benefit is supposed higher quality, the cost is extra effort that would not otherwise have had to be expended.

The approach described in the next section attempts to provide some insight into the quality of a software product without requiring the implementor to expend effort over and above the effort that would normally be expended to write requirements, create tests and monitor defects.

3 A PRAGMATIC APPROACH

This section provides an overview of a pragmatic approach to gaining insight into quality in software development. More details about the approach are covered in subsequent sections.

A software development project can be defined as a project undertaken to provide some software product that fulfills some requirements of a user or set of users [2]. As discussed in the literature covered in Section 2, *one* of the indicators of quality in a software development project is the degree to which the produced software fulfills the stated requirements of the user. Such a view of quality corresponds to the *manufacturing-based* approach to quality mentioned in the introductory remarks of Section 1. Code is developed in order to meet these requirements, and tests are run against the code to confirm whether requirements have been met. In cases where tests fail (indicating a requirement has not been met) a defect is raised. The aim of a software project, then, is to fulfil all of the requirements and confirm these requirements are met, handling defects as they arise. In general, the approaches discussed in the Section 2 have some view of requirements, tests and defects and the relationship between them.

The main difference between the approaches discussion in Section 2 and the approach discussed in this Section is the effort required to gather and calculate metrics associated with quality. In this approach, metrics pertaining quality are calculated automatically based on processes and information that would normally be part of the development process anyway (tests, requirements and defects) so relatively little effort is required to implement this approach.

In the text below, a metric or product quality indicator will be proposed to measure the extent to which stated requirements have been met. To this extent, the approach is also classifiable as a *product-based* approach to quality—i.e. it holds that quality is measurable and related to the degree to which the product possesses attributes that are specified as requirements. However, it should be emphasised that the measure is not being proposed as a panacea to the multi-dimensional problem of assessing quality, but rather as a single and pragmatic mechanism to assist in addressing that problem.

3.1 Assumptions

A software project consists of a set of deliverables. Among these deliverables are code and documentation. The delivered code and documentation are created from some source code (and documentation) that are developed over the course of the project by software developers. The source code is represented by a set of files. Over the course of the project, these source files will be created, modified and tested until they fulfil the stated requirements.

Most software development methodologies require a testing phase, so at certain points of the software development life cycle, tests will be run against an interim software product, or *release*. The execution of these tests may uncover differences between the stated requirements and what the interim product actually delivers. These differences can be handled as defects. Successive phases or iterations of the development life cycle, then, will attempt to address these defects, with the goal being no deviation between the product and the requirements (i.e. there are no defects).

The assumptions used in this approach are listed below.

- There is a set of requirements indicating the requirements that the developers of the software need to fulfill. The requirements are easily identifiable and can be enumerated and can change over the course of the project. The requirements can be functional or non-functional, but this approach assumes that:
  - the requirements (functional and non-functional) have been stated in some set of requirements, and
  - all of the stated requirements can be verified using some test.
- There is a set of source files that are developed to fulfil the requirements of the software product. Each source file contributes to the fulfilment of zero or more requirements. Some source files may provide helper functions such as string formatting and are therefore not directly involved in fulfilling requirements. A requirement may be fulfilled by a single source file or a number of source files.

- There is a set of tests that are run against a set of source files to verify that there is no deviation between what is delivered and what is required (i.e. there are no defects). A test exposes only one defect and a defect is raised by only one test. This does imply some constraint on how tests are designed.
- Each requirement will be verified by one or more test(s). Because source files are associated with requirements, this implies that a set of tests is associated with a set of source files—the cardinality of the relationship being unrestricted.
- There is a set of defects, which represent the deviation between delivered product and stated requirements. Each defect is found as a result of a test execution. Each source file may be associated with zero or more defect(s) and each defect may be associated with one or more file(s). As noted by Dijkstra, "Program testing can be used to show the presence of bugs, but never to show their absence" [21], and the approach proposed here only deals with those defects that are observable as a result of running tests on the software. If there are latent defects that are not uncovered by testing (for example, because the testing was not rigorous enough), then these defects would not be considered in this approach.

3.2 The Product Quality Indicator

It is generally considered good practice for software producers to use some form of SCM system in order to manage changes to their software as the software development lifecycle progresses. Subversion [22] is a popular SCM system and supports the notion of executing arbitrary processes when certain events occur, such as a change to some file under the control of the repository. The approach proposed here uses this mechanism to automatically calculate values for metrics as the code for the project is modified and to store the values of the metrics with the source code in the SCM repository. The values are used to provide an insight into the quality of the product under development. The benefit of this approach is that it uses processes and concepts that are generally part of the software development process.

The metrics proposed here are defined based on metrics intrinsic to most software development projects: the proportion of the number of requirements met to the total number of requirements and the proportion of the number of tests that pass to the total number of tests. The definitions of these metrics are as follows.

$$R = \frac{R_{met}}{R_{total}} \tag{1}$$

where R_{met} and R_{total} are the number of requirements claimed to have been met and the total number of requirements, respectively. The concept of using the proportion of total requirements to the requirements met or defects is also discussed in [23], [24] and [25].

$$T = \frac{T_{run} - D}{T_{total}} \tag{2}$$

where T_{run} is the number of tests run (pass or fail), T_{total} is the total number of tests defined, and D is the total number of defects raised by the tests that have run. Each test can give rise to zero or one defects.

R provides an indication of the completeness of the project. T provides an indication of how much the value of R is validated by tests that pass; a confidence factor of sorts. The value of R could be high because all of the requirements are stated as met, but without the tests to corroborate that claim, the quality cannot be said to be high. These metrics are combined to give a view of quality, Q, as below. The concept of using the proportion of tests passing to failing is discussed in [26] and [27].

$$Q = \alpha \times R + (1 - \alpha) \times T \tag{3}$$

where α is the weight of the value of R and $0 < \alpha < 0.5$. The value of α is less than 0.5 because the relative importance of T is higher as it provides a confidence factor for the value of R.

As indicated in Equation 3, Q is calculated as a linear combination of R and T where the sum of the weights (or co-efficients) is 1. The reason for this is so that different importance might be placed on each of R and T. For example, a relatively low value of α would ensure that Q remains realistic in a context where a high proportion of requirements have been met but a low proportion of tests passing have been passed. Theoretically, the value of α is between 0 and 1. Here however α is constrained between 0 and 0.5, so that the value of T is given a higher weight than the value of R in the calculation of Q because the weighted value of R will always be less than 0.5. The values of R and T range between 0 and 1 and hence Q ranges between 0 and 1.

The next section describes *Metaversion*, an implementation of the approach described here using Subversion as an SCM repository.

4 METAVERSION

This section describes an implementation of the approach described in Section 3 called *Metaversion*.

4.1 Collecting source file meta-data

As mentioned previously in this section, for each of the source files in a software project, the following information must be stored (as per the assumptions noted in Section 3.1):

- details of the tests that include or act upon the source file;
- details of the defects that the source file participate in—or contribute to; and
- details of the requirements that are claimed (and are proven using tests) to be met by the source file.

It should be borne in mind that there may need to be significant effort expended in order to determine which file gives rise to a defect, but this effort would have to be expended in the course of fixing the defect anyway.

Using Subversion, which has meta-data capabilities using *properties*, the required meta-data can be stored. The approach discussed here does not rely on Subversion being used as the SCM system. The approach would work with any SCM system that supports setting arbitrary metadata on individual files in the repository. Subversion was chosen for this implementation, although there are other SCM systems such as Git [28] that could be used.

Subversion properties can have any name and are version-controlled along with the file itself. This means that over time, as the contents of the file change and requirements, tests and defects are associated with it, the values of the properties also change. The property names and values that are to be set against each version of a source file (as applicable) are listed below. Note that the identifiers referred to below are the test numbers, requirements numbers or defect numbers that identify each of the tests, requirements or defects in the project.

mv:tests-pass A list of identifiers for the tests that have run and are known to have passed for this version of the file.

mv:tests-total A list of identifiers for all of the tests that are associated with this version of the file.

mv:tests-fail A list of identifiers for the tests that have run and are known to have failed for this version of the file.

mv:defects-open A list of identifiers for the defects that have been reported against this version of the file and are as yet unresolved.

mv:defects-closed A list of identifiers for the defects that have been reported against this version of the file and are resolved.

mv:req-met A list of identifiers for requirements that have been proven to be fulfilled by this version of the file.

mv:req-unmet A list of identifiers for requirements that have been proven not to be fulfilled by this version of the file (e.g. after a defect has been raised against a previously met requirement).

mv:req-total A list of identifiers for requirements that this version of the file is supposed to fulfil.

The *mv:req-total* property would be set based on the requirements specification—hence the assumption in Section 3.1 that, for ease of reference, there should be an enumerated list of requirements. Based on these requirements, a set of tests would be created, and the *mv:tests-total* property would be set from these.

If testing were an automated process, the setting of the *mv:tests-pass* and *mv:tests-fail* properties could be done by the software tool performing the testing. If a defect management system was in use, when a defect is raised, it could set the value of the *mv:defects-open* property. Similarly, when a defect was resolved, the *mv:defects-closed* property could be set. The *mv:req-met* and *mv:req-total* properties could have their values automatically set by some requirements management tool, if one were in use. If there are details about which tests are related to which requirements, then details about met and unmet requirements can be inferred from passed and failed tests respectively. The properties *mv:req-unmet* and *mv:tests-fail* are used to store this information.

If no automated software is in use for these areas of the software project, the values could be set manually as part of the planning (for requirements and tests) and development and testing (for tests and defects).

Figure 1 depicts the collection of the properties from the various sources in the software project.

4.2 Using source file meta-data

The value of storing the property meta-data is in the analysis thereof. Using the values of these properties and the quality equation described in Equation 3 (and any other metrics that can be derived from these properties), it is possible to get some idea of the quality of a particular source file, but perhaps of more value is the collective quality of the set of source files: the quality of the software product as a whole. This approach (as mentioned before) is not prescriptive regarding the mechanism for extracting the meta-data nor the tools to use to do it. This implementation is a suggestion of how this could be done.

One approach to using the source file meta-data would be to step through every file in the repository, read the values of the properties and build a report from that data. As the size of the repository grows, however, this approach would take longer to produce results because there are more source files for which to store meta-data and hence more meta-data.

A more scalable approach would be to have the Subversion repository react when a property is set on a file (and that property is one of the properties mentioned in Section 4.1) by storing the data in some intermediate location. The Subversion application provides this event-based mechanism through the use of *hooks*. These hooks allow custom applications to run when some activity occurs in the repository. The hook of interest to Metaversion is the *post-commit* hook. This hook is triggered when any file or change to a file is committed to the repository (including property changes). These hooks allow for an event-driven approach to the analysis of the meta-data as opposed to a polling approach. The data needed to perform analysis and reporting will be built up over time and the reporting application need only consult the intermediate store of data to build reports and not the entire Subversion repository.

Figure 2 shows how the properties set on source files in the repository will be stored in an intermediate database. The post-commit hook will be configured to call an application. This application is called *MetaversionCommitHandler*, and was produced as part of this study.

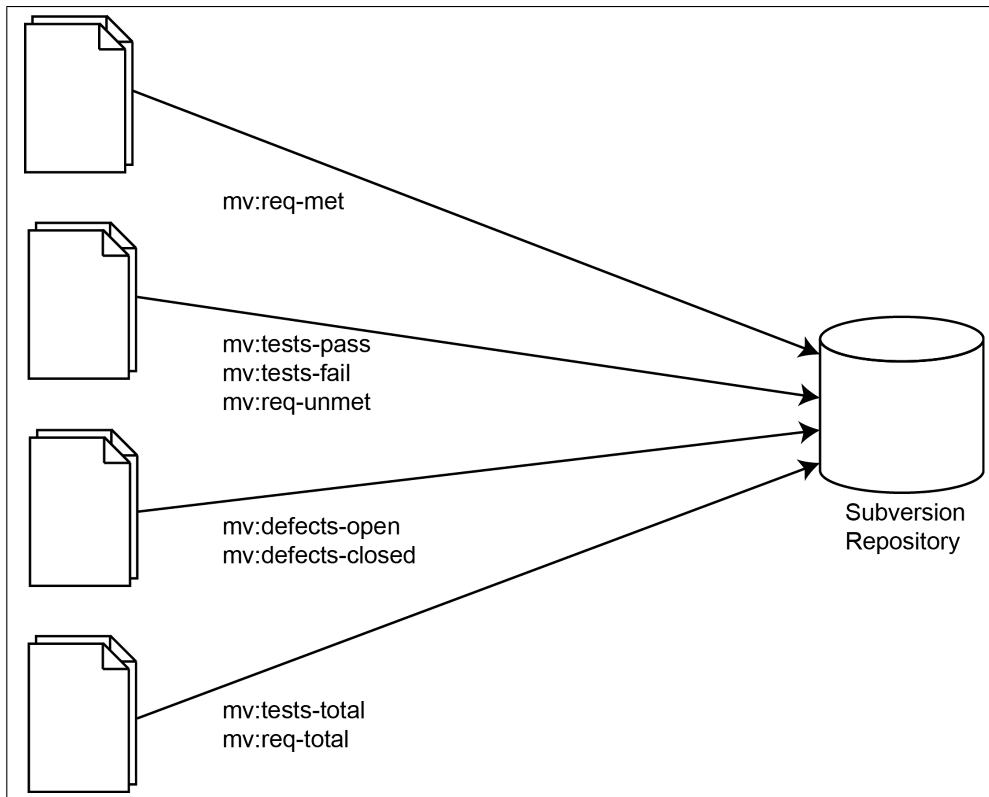

Figure 1: Gathering source file meta-data

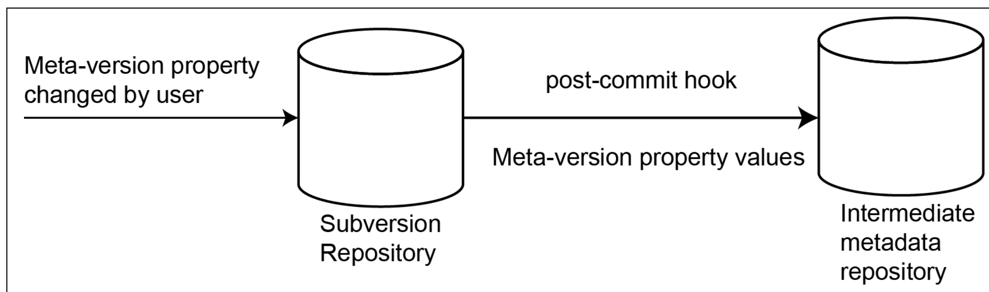

Figure 2: Reacting to post-commit hook

The meta-data stored with source files will be extracted using a reporting application. This reporting application will gather the meta-data from the repository and produce reports that will aid the development team, managers and sponsors of the software product to gain insight into the quality performance of the software. This reporting application was called *MetaversionReporter* and was produced as part of this study.

4.3 Metaversion Data Model

Metaversion reflects the relationship between tests, defects and requirements as indicated in Figure 3. Each file can have zero or more tests, zero or more defects and zero or more requirements associated with it.

4.4 MetaversionCommitHandler

When a user (developer, manager, sponsor, etc.) sets a property on a file in the repository and commits the changes to the Subversion repository, the repository invokes the post-commit hook. The hook, in turn, invokes the *MetaversionCommitHandler* application. The *MetaversionCommitHandler* application calculates which properties have changed (in particular, the properties from the list in Section 4.1 for each file for which properties have been set). A MySQL [29] database is updated with the details stored in the properties: requirements, tests and defects are associated with files, as appropriate. Figure 4 shows this process. *MetaversionCommitHandler* is used in the case study in Section 5.

4.5 MetaversionReporter

As mentioned, the value of the meta-data stored in the intermediate database is in the analysis thereof. The *MetaversionReporter* application is run by a member of the team developing the software in order to gain insight into the quality of the product. (Here no specific position is taken about whether this role should be

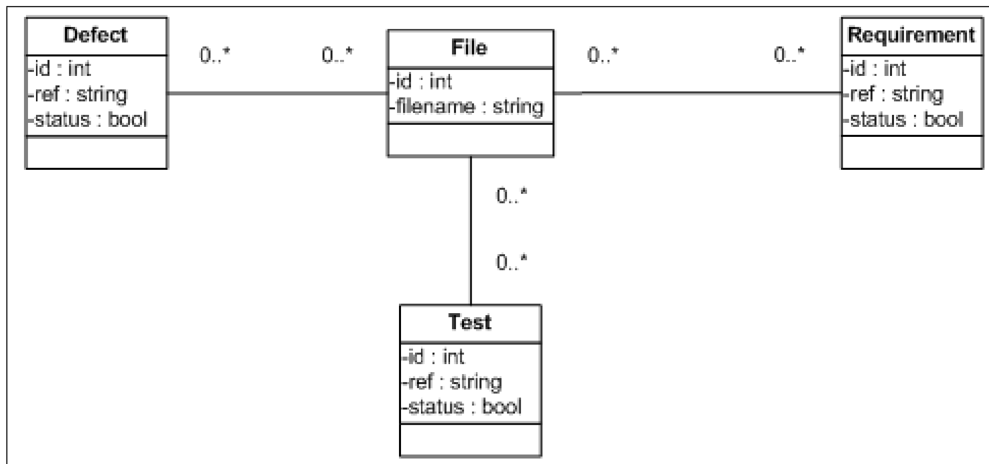

Figure 3: Metaversion data model

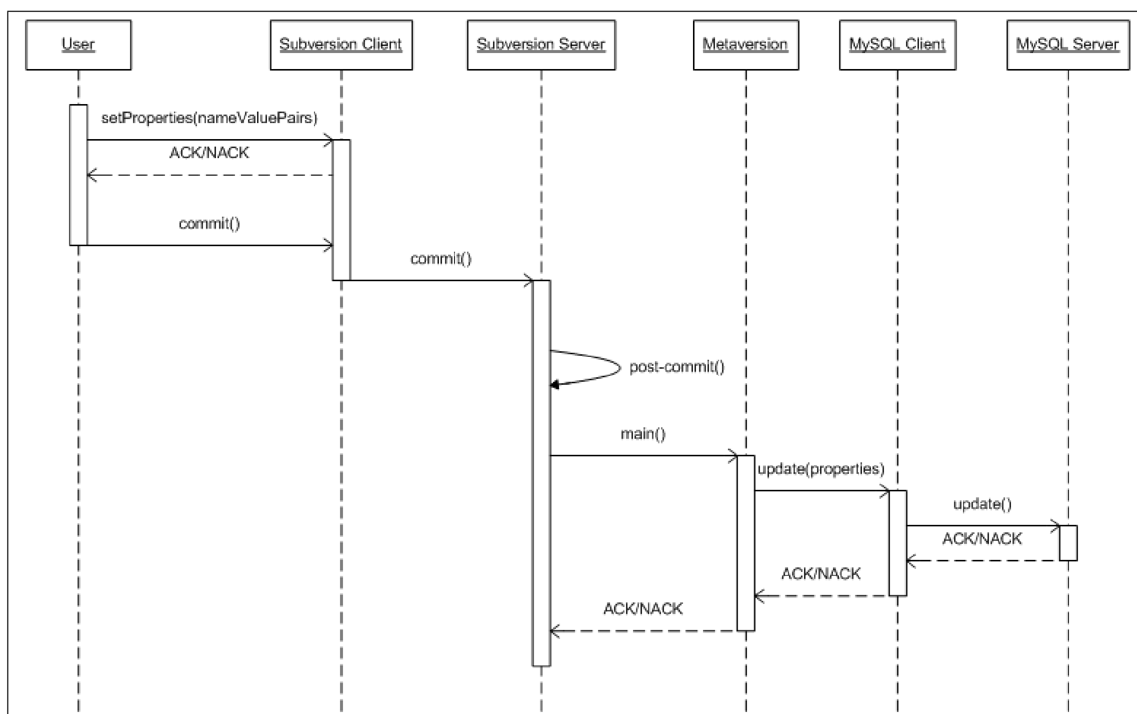

Figure 4: Metaversion commit process

assigned to the developer, the manager, the sponsor, etc.) The *MetaversionReporter* application reads the information from the MySQL database that is populated by the *MetaversionCommitHandler* application. This information is then used to calculate the value of Q (as defined in Equation 3). The information is then displayed to the user. This process is depicted in Figure 5. *MetaversionReporter* is used in the case study in Section 5.

The next section presents a case study where this approach was implemented using the implementation described in this section.

5 CASE STUDY

This section describes a small case study that was performed using an implementation, called Metaversion, of the approach described in Section 4. The value of Q as defined in Section 4 is provided across the 6 interim products that the project yielded. For all calculations of Q, a value of 0.3 was used for α—i.e. the proportion of tests run and consequent defects exposed was weighted more heavily than the proportion of requirements met. A survey was used as a comparison against the values of Q that were automatically calculated after each interim product.

The case study was performed on a software project called Timezonr[1], which is an Android [30] application. The application is a small application of around 1500 lines of code. The application was written by a single developer—the first author of this paper. It was tested by two independent people—i.e. non-authors of the paper. The testers answered the survey questionnaire after each interim build, and from this questionnaire,

[1] The TimeZonr application can be found at `http://lnkr.co.za/timezonr`.

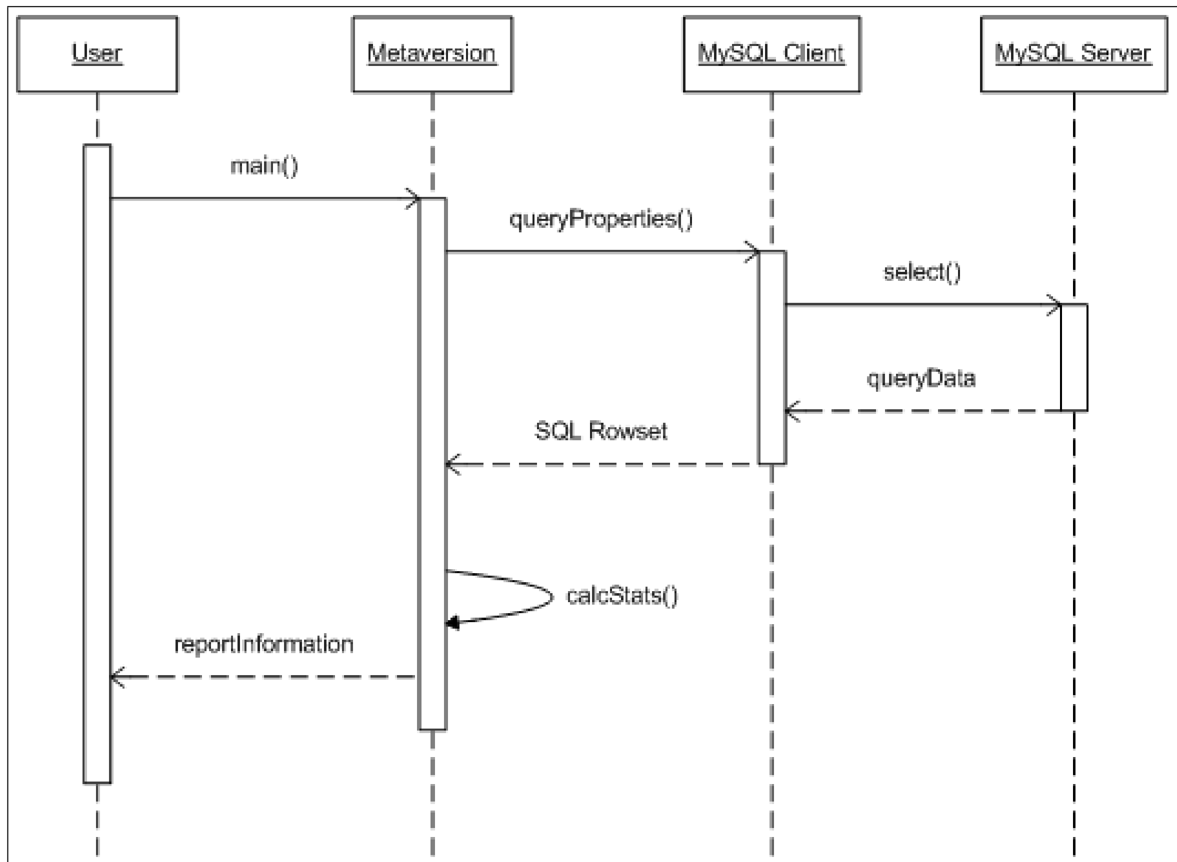

Figure 5: Metaversion reporting process

an indication of 'perceived quality' was found. This 'perceived quality' was deduced, along with an indication of the completeness of the product and the confidence that the respondents of the survey had in their perception of completeness. These responses were compared with the calculated value for Q, R and T, respectively.

Figures 6, 7 and 8 show the values for Q, R and T, respectively, as calculated by the approach, against the average responses of the survey.

Figure 6 indicates that the perceived quality increased as the product was developed. This implies that the perception of the testers was that as the project was being developed, the product produced got closer to what they wanted. The calculated quality Q also increased as the product was being developed. This is because, as the product was developed, more of the requirements were being met, and more of the tests indicating adherence to requirements were passing.

Figure 7 indicates how the completeness of the product was perceived by the testers and the calculated values of completeness. In general, both perceived and calculated completeness increased over the course of the project, and this indicates that testers believed that the project was progressing towards a state of completion, and that the number of requirements (which were known to the testers) being met was increasing over the course of the project. At build 4 and 5, the perceived completeness was less than the calculated completeness. This could well be because of the combination of the testers' understanding of the term *completeness* and the discrete interval from which they had to choose values for completeness. The calculated and perceived values were only equal once other than the start and end of the project. At build 4, the calculated completeness was 1.0, which implies that all requirements were claimed to have been met. The perception of the respondents, however, was that the project was not complete. Since the respondents knew the requirements, this difference could be attributed to the presence of defects. The respondents knew that the software had defects, and they may have surmised therefore that the software was incomplete.

Figure 8 indicates that the confidence in completeness increased as the project progressed. The testers perceived that they were more sure of their assessment of completion as the project progressed. This could be due to familiarity with the product, and an increase in understanding of how the application worked. In addition to this, this measure indicates that the testers felt they had enough of a view of the completeness of the product to be sure of it. For the calculated confidence in completion, this indicates that a higher proportion of the tests that were being run were passing, which lends some degree of confidence to the completeness of the product. The values of the completeness confidence from the respondents were all high. This could be attributed to two things. Firstly, the responses only had discrete values, and so the respondent would not have been able to give an answer inbetween the values. Thus, they may have, for example, chosen 4 out of 5, when they actually wished to give a response of 3.5.

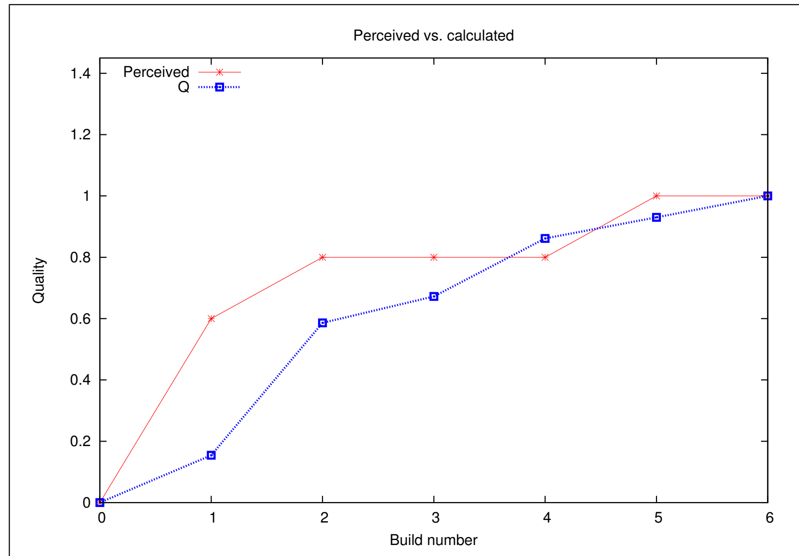

Figure 6: Quality: perceived vs. calculated

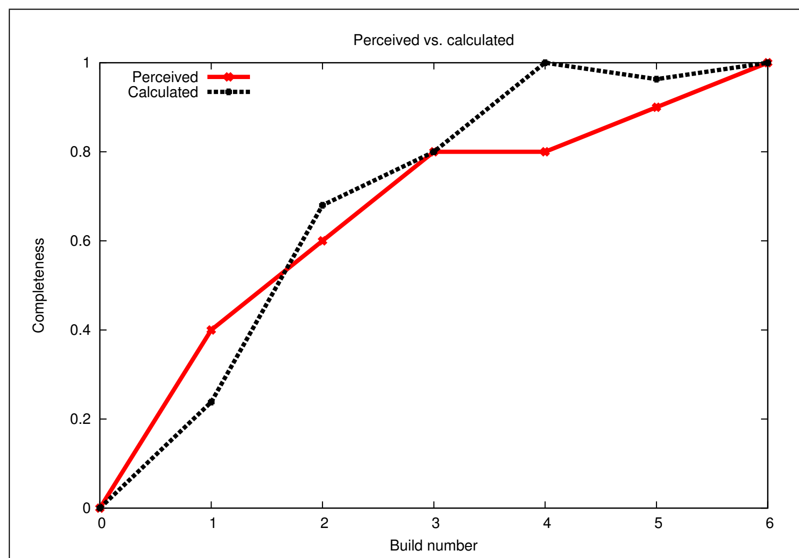

Figure 7: Completeness: perceived vs. calculated

Secondly, the respondents knew the requirements of the build when they were testing, so they could actually prove their answer to the completeness question. Since all of the responses were not 5 out of 5, this could imply they did not do this, or it could imply that there was some other factor influencing their understanding of the question.

These figures show that, in this exercise, the 'perceived quality' of the product had a fairly close similarity to the metric calculated by the new approach. This supports the view that the metric Q could be used to gain some insight into the quality of the software product.

6 CONCLUSION

This paper provided an overview of approaches to gaining insight into the quality of software products and pointed to the fact that there appears to be a deficit in quality metrics that can be easily and pragmatically gathered. A pragmatic approach was proposed in Section 3 to provide software producers with insight into product quality based on processes, systems and artifacts usually present in a software project. The case study described in Section 5 is seen as a limited pilot study to test the feasibility of implementing the proposed approach *in situ*.

The case study is clearly limited in terms of project size, number of testers and the self-involvement of one of the authors. It therefore does not claim to show universal applicability of the approach proposed. Nevertheless, it does provided evidence that quality as perceived by non-developers (one of the testers was actually also the client of the product) coheres with the proposed metrics. Of course, more evidence as to the level of insight the approach provides into software quality should be attained by implementing the approach across a wider variety of projects of varying sizes.

Executing the implementation of this approach

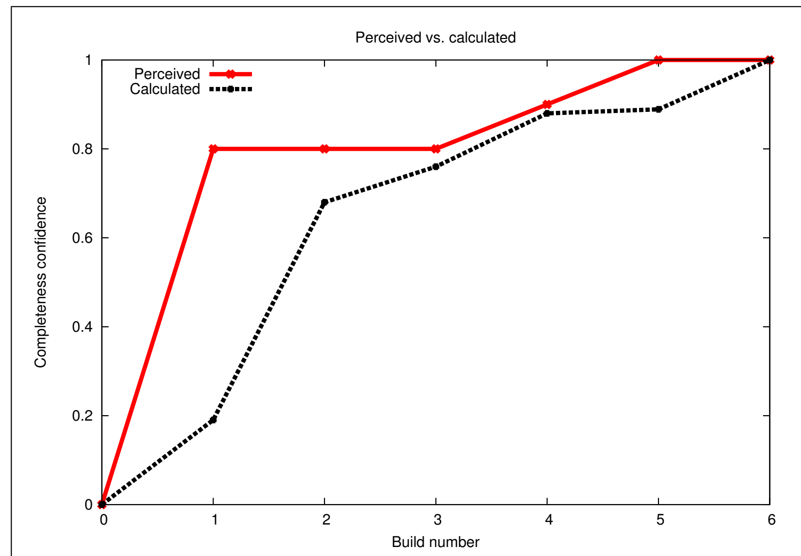

Figure 8: Completeness confidence: perceived vs. calculated

also brought to light further possibilities for future work to improve the approach. The most immediate are as follows:

- As stated, the approach requires that the implementor sets requirements, tests and defects against each file in the repository. This approach still requires effort on the part of the implementor. Future work could be done to automate this process, by integrating requirements gathering software and defect tracking software. Such integrations do exist, for example Scmbug [31].
- Currently, the approach proposes only three metrics, and further work could be done to provide further metrics based on the SCM repository. An example could be productivity (perhaps based on the number of lines of code submitted per day).
- Currently, the approach proposes Subversion as an SCM, and future work could investigate making the approach more generic so that any SCM could be used.

From a broader perspective, the study highlights the possibilities of using version control systems more effectively for gathering statistical evidence to monitoring quality as software projects progress through their life-cycles.

REFERENCES

[1] IEEE Computer Society. "IEEE standard glossary of software engineering terminology", 1990.

[2] International Organization for Standardization. *SANS 9000:2005 — Quality management systems — Fundamentals and vocabulary*. Standards South Africa, 2005. ISBN 0-626-17567-4.

[3] IEEE Computer Society. "IEEE standard for a software quality metrics methodology", 2004.

[4] D. A. Garvin. "What does "product quality" really mean?" *Sloan Management Review*, vol. 26, no. 1, pp. 25–45, 1984.

[5] M. B. Chrissis, M. Konrad and S. Shrum. *CMMI: Guidelines for Process Integration and Product Improvement*. Addison Wesley Professional, 2003. ISBN 978-0-321-15496-5.

[6] W. S. Humphrey. "The Personal Software Process (PSP)". Tech. rep., Carnegie Mellon Software Engineering Institute, 2000.

[7] International Organization for Standardization. *Software Engineering — Software product Quality Requirements and Evaluation (SQuaRE) — Guide to SQuaRE*. Standards South Africa, 2007. ISBN 978-0-626-19201-3.

[8] B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. J. Macleod and M. J. Merrit. *Characteristics of software quality*. TRW Series of Software Technology. North-Holland Publishing Company, 1978. ISBN 0444851054.

[9] L. E. Hyatt and L. H. Rosenburg. "A software quality model and metrics for identifying project risks and assessing software quality". In *European Space Agency Software Assurance Symposium and the 8th Annual Software Technology Conference, Product Assurance Symposium and Software Product Assurance Workshop, Proceedings of the meetings held 19-21 March, 1996 at ESTEC, Noordwijk, the Netherlands*, p. 209. 1996.

[10] J. A. McCall, P. F. Richards and G. F. Walters. "Factors in software quality". Tech. rep., Rome Air Development Center, Air Force Systems Command, Griffiss Air Force Base, New York, 1977.

[11] H. D. Mills, M. G. Dyer and R. C. Linger. "Cleanroom software engineering". *IEEE Software*, vol. 9, pp. 19–25, 1987.

[12] H. Barkmann, R. Lincke and W. Löwe. "Quantitative evaluation of software quality metrics in open-source projects". In *International Conference on Advanced Information Networking and Applications Workshops*, pp. 1067–1072. 2009.

[13] G. Gousios and D. Spinellis. "Alitheia Core: An extensible software quality monitoring platform". In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pp. 579–582. IEEE

Computer Society, Washington, DC, USA, 2009. ISBN 978-1-4244-3453-4.

[14] P. A. Currit, M. G. Dyer and H. D. Mills. "Certifying the Reliability of Software". *IEEE Transactions on Software Engineering*, vol. 12, no. 1, pp. 3–11, 1986.

[15] International Organization for Standardization. "SANS 9126:2003 — Software Engineering — Product Quality", 2003.

[16] International Organization for Standardization. *Software engineering - Product quality Part 1: Quality model*. Standards South Africa, 2003. ISBN 0-626-14674-7.

[17] B. Kitchenham and S. Pfleeger. "Software quality: The elusive target". *IEEE Software*, vol. 13, no. 1, pp. 12–21, 1996.

[18] D. Stelzer, W. Mellis and G. Herzwurm. "A critical look at ISO 9000 for software quality management". *Software Quality Control*, vol. 6, no. 2, pp. 65–79, Oct. 1997. ISSN 0963-9314.

[19] M. Staples, M. Niazi, R. Jeffery, A. Abrahams, P. Byatt and R. Murphy. "An exploratory study of why organizations do not adopt CMMI". *Journal of Systems Software*, vol. 80, no. 6, pp. 883–895, Jun. 2007. ISSN 0164-1212.

[20] N. Khurshid, P. L. Bannerman and M. Staples. "Overcoming the first hurdle: Why organizations do not adopt CMMI". In *Proceedings of the International Conference on Software Process: Trustworthy Software Development Processes*, ICSP '09, pp. 38–49. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-01679-0.

[21] P. Naur and E. B. Randell (editors). *Software Engineering: Report of a conference sponsored by the NATO Science Committee*. 1961.

[22] M. Mason. *Pragmatic version control using Subversion, 2nd edition*. The Pragmatic Bookshelf, 2006. ISBN 978-0-9776166-5-7.

[23] V. Wyatt, J. Distefano, M. Chapman and E. Aycoth. "A metrics based approach for identifying requirements risks". In *28th Annual NASA Goddard Software Engineering Workshop*. 2003.

[24] C. Michael, G. McGraw and M. Schatz. "Generating software test data by evolution". *Software Engineering, IEEE Transactions on*, vol. 27, no. 12, pp. 1085–1110, 2001. ISSN 0098-5589. doi:10.1109/32.988709.

[25] D. Richter. *Determination of concurrent software engineering use in the United States*. Universal Publishers, 1999. ISBN 9781581120653. URL `http://books.google.co.za/books?id=lLXWAa0VFLMC`.

[26] A. Vetro, M. Morisio and M. Torchiano. "An empirical validation of Find Bugs issues related to defects". In *15th Annual Conference on Evaluation & Assessment in Software Engineering*. 2011.

[27] M. Cohn and D. Ford. "Introducing an agile process to an organization [software development]". *Computer*, vol. 36, no. 6, pp. 74–78, 2003.

[28] T. Swicegood. *Pragmatic version control using Git*. The Pragmatic Bookshelf, 2008. ISBN 978-1-9343561-5-9.

[29] P. DuBois, S. Hinz, J. Stephens, M. Brown and T. Bedford. *MySQL 5.1 reference manual*. Oracle Corporation, 2008.

[30] Google. "Android". URL `http://www.android.com/`.

[31] K. Makris. "Scmbug manual", 2004. URL `http://lnkr.co.za/scmbug-manual`.