The EMISQ method and its tool support-expert-based evaluation of internal software quality

R. Plösch · H. Gruber · A. Hentschel · Ch. Körner · G. Pomberger · S. Schiffer · M. Saft · S. Storck

Abstract There is empirical evidence that internal software quality, e.g., the quality of source code, has great impact on the overall quality of software. Besides well-known manual inspection and review techniques for source code, more recent approaches utilize tool-based static code analysis for the evaluation of internal software quality. Despite the high potential of code analyzers the application of tools alone cannot replace well-founded expert opinion. Knowledge, experience and fair judgment are indispensable for a valid, reliable quality assessment, which is accepted by software developers and managers. The EMISQ method (Evaluation Method for Internal Software Quality), guides the assessment process for all stakeholders of an evaluation project. The method is supported by the Software Product Quality Reporter (SPQR), a tool which assists evaluators with their analysis and rating tasks and provides support for generating code quality reports. The application of SPQR has already proved its usefulness in various code assessment projects around the world. This paper introduces the EMISQ method and describes the tool support needed for an efficient and effective evaluation of internal software quality.

1 Introduction and motivation

Commonly, the general term software quality combines external and internal software quality. The standard ISO 9126 [1] defines software quality as “the totality of characteristics of a software product that bear on its ability to satisfy stated and implied needs”. However, this general definition of software quality does not suffice for practical application. With the help of quality models, the concept of quality is operationalized by deriving subconcepts, with external and internal software quality at the highest level of abstraction.

**External software quality** covers those quality aspects that are noticeable by the users of a software product. Typical examples of external quality attributes are *usability* and *functionality*. Assessing the external quality of a software product requires an operational product that has to be executed in order to be evaluated.

**Internal software quality** covers quality aspects that are primarily visible to software architects, developers and testers; it focuses on the source code, internal documentation, architectural descriptions, etc. Even if users do not directly get in touch with internal software quality, it is an intuitive observation that internal quality affects external quality to a high degree. This observation is also supported by
several studies (e.g., [2–4]) though Fenton and Neil found some principal problems with such surveys [5].

On a technical level assessing the internal quality of software allows one to answer a number of questions: What are the reasons for specific software-related problems? Is the software future-proof? Which improvement and refactoring activities bear the greatest potential? Are the craftsmanship and technology used appropriately? Answering these questions is crucial for go/no-go decisions at important milestones of a software project or for preparing strategic decisions like platform selection or software acquisition. Furthermore the internal software quality is a good indicator for making software aging visible and for identifying maintenance needs.

This paper has the focus on the assessment of internal quality of software by means of tool-based static code analysis, accompanied by an expert judgment of the static code analysis results. However, static code analysis is not sufficient to answer all the questions mentioned above. Therefore, besides static code analysis other methods like architectural reviews and tests also provide valuable input for judging the internal software quality. As far as source code is concerned, methods and tools for dynamic code analysis (e.g., debugging and profiling) help to detect defects like logically incorrect code, memory leaks and dead locks. Such methods are not described in this paper although they are considered by our approach.

Investigating the source code by static code analysis can be either performed manually or by using tools. Manual analysis methods like code inspections (as proposed by [6,7]) are commonly accepted methods to identify quality-related problems. Runeson et al. [8] performed a meta-analysis of various experiments regarding the effectiveness and efficiency of commonly known inspection techniques. As a conclusion the effectiveness as well as the efficiency of manual code inspection techniques is rather low—0.82 detected defects per hour in a commercial setting with communication overhead, complex code fragments, etc. For higher efficiency the evaluation of large software systems (more than 100,000 lines of code) requires the use of code analysis tools. Such tools (e.g., PC-Lint [9] for C/C++, FxCop [10] for C# or PMD [11] for Java) typically compute metrics and try to directly detect code anomalies.

Many well-known metrics exist that measure the size and structure of software products (e.g., “lines of code”, “cyclo-matic complexity” or “fan-in/fan-out”) or object-oriented issues (e.g., “Metrics Suite for Object Oriented Design” [12]) and associate threshold values with each metric which should not be exceeded by well implemented software. Although such metrics are supposed to contribute significantly to the determination of quality attributes, we are convinced—based on our experience as assessors—that only a limited aspect of internal software quality can be captured by applying those metrics to state of the art software products. Our opinion is supported by several studies (e.g., [13,14]) that show weak explanatory and prediction power of particular metrics with respect to error-proneness, maintenance costs and other quality issues. Due to the mostly low significance of metrics as mentioned above a common approach of current tools for static code analysis is to identify and count quality breaking source code spots [15]. The usual goal is to get as much qualitative and quantitative information as possible by deeply checking the source code in order to find suspect patterns (bad smells), programming errors, violations of programming conventions etc. What to search for is typically provided by so-called rules, which represent (configurable) programming conventions and/or expert knowledge on what to obey and what to avoid when writing source code. With the data provided by code analysis tools at hand, a rough estimation of the overall quality of software as well as the identification of subsystems or components with probably minor quality is possible.

Despite the insights, which can be obtained by automatically identifying and counting threshold and rule violations, in almost all cases, it is nearly impossible to automatically transform tool output into a meaningful assessment of internal software quality. There are several reasons, why this is difficult to do, with the following most prominent causes:

- Whether the violation of a rule or an exceeded value of a metric has to be classified as a quality issue cannot be always determined automatically. In many cases it depends on the specific context or on the application domain if and how quality is affected. For example, a method that dispatches user interface events to event handlers usually exceeds length and complexity thresholds, as the dispatching method typically has one if- or switch-branch for each event and can be therefore very long and/or deeply nested. However, in that particular case that is not a problem at all but is a well understood coding practice.

- The calculation of metrics as well as the detection of rule violations is error-prone. The dominating aspect in this context is the detection of so-called false-positives, i.e., violations or metric values that are supposed to indicate a quality problem but were falsely detected or calculated. In a recent survey, taking three tools into account, Wagner et al. [16] found out that two thirds of rule violations were falsely detected.

Besides these more technical issues it is also important to understand the influence of a violation for a certain quality attribute or the effort necessary to fix the problem. Thus tool output can give a first hint, but an expert judgment is necessary to meaningfully assess the internal software quality.

The Evaluation Method for Internal Software Quality (EMISQ) is a systematic method for expert-based evaluation of internal software quality. The method relies on static code