Software Qual J (2012) 20:287–307
DOI 10.1007/s11219-011-9144-9

Standardized code quality benchmarking for improving software maintainability

Robert Baggen · José Pedro Correia · Katrin Schill · Joost Visser

Abstract We provide an overview of the approach developed by the Software Improvement Group for code analysis and quality consulting focused on software maintainability. The approach uses a standardized measurement model based on the ISO/IEC 9126 definition of maintainability and source code metrics. Procedural standardization in evaluation projects further enhances the comparability of results. Individual assessments are stored in a repository that allows any system at hand to be compared to the industry-wide state of the art in code quality and maintainability. When a minimum level of software maintainability is reached, the certification body of TÜV Informationstechnik GmbH issues a Trusted Product Maintainability certificate for the software product.

Keywords Software product quality · Benchmarking · Certification · Standardization

1 Introduction

The technical quality of source code (how well written it is) is an important determinant for software maintainability. When a change is needed in the software, the quality of its source code has an impact on how easy it is: (1) to determine where and how that change can be performed; (2) to implement that change; (3) to avoid unexpected effects of that change; and (4) to validate the changes performed.

R. Baggen · K. Schill
TÜV Informationstechnik GmbH, Essen, Germany
e-mail: r.baggen@tuvit.de

K. Schill
e-mail: k.schill@tuvit.de

J. P. Correia (✉) · J. Visser
Software Improvement Group, Amsterdam, The Netherlands
e-mail: j.p.correia@sig.eu

J. Visser
e-mail: j.visser@sig.eu

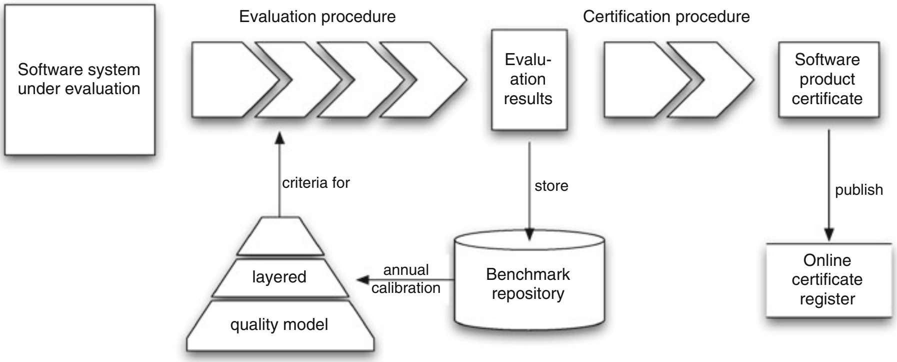


Fig. 1 Evaluation, benchmarking and certification of software product quality

However, many projects fail to assess code quality and to control it the same way as the other classical project management KPIs for timeline or budget. This is often due to the fact that projects lack a standardized frame of reference when working with source code measurements. As a result, the quality of the product remains unknown until the (final) testing and problem fixing phase begins.

In this paper, we describe an approach developed by the Software Improvement Group (SIG) for code analysis and quality consulting focused on software maintainability. The approach uses a standardized measurement procedure based on the ISO/IEC 9126 definition of maintainability and source code metrics. Measurement standardization greatly facilitates the collection of individual assessments in a structured repository. Based on the repository, any system at hand can be compared to the industry-wide state of the art in code quality and maintainability. Procedural standardization in evaluation projects further enhances the comparability of results. When a minimum level of software maintainability is reached, TÜV Informationstechnik GmbH (TÜViT) issues a Trusted Product Maintainability certificate for the software product. An illustration of the approach is provided in Fig. 1.

This paper is organized as follows. In Sect. 2, we start with an explanation of the measurement model and its calibration against a benchmark database of measurement results. In Sect. 3, we describe the standardized evaluation procedure in which the model is used to arrive at quality judgments in an evaluator-independent manner. This evaluation procedure is used as part of the software product certification scheme and software quality consulting services, as explained in Sect. 4 Examples of the usefulness of the standardized approach in practice are presented in Sect. 5. In Sect. 6, we approach some potential points of criticism to our approach. Section 7 discusses related work. Finally, in Sect. 8, we present some concluding remarks and directions for future work.

2 Measuring software via code metrics

The application of objective metrics for the measurement and improvement of code quality has a tradition of more than 40 years. Today, code measurement is seen as pragmatic work—the goal is to find the right indicator for a given quality aspect and a given development environment. However, being too creative about the measurements may preclude helpful comparisons with other projects. Therefore, metrics have to be chosen with clear reference to an agreed standard—e.g. the ISO/IEC 9126 international standard for software product quality (International Organization for Standardization 2001).

In the following subsections, we summarize the measurement model and its calibration against a benchmark repository. Full details can be found in the previous publications on the design of the model (Heitlager et al. 2007), its calibration (Alves et al. 2010), its evaluation (Correia et al. 2009; Luijten and Visser 2010; Bijlsma 2010), its application to open-source software (Correia and Visser 2008b), and the underlying benchmark (Correia and Visser 2008a). For self-containment, some details are also available in Appendix.

2.1 Software code metrics for maintainability

Conceiving maintainability as a function of code quality leads to a number of code metrics as candidates in maintainability assessments. SIG chose 6 source code properties as key metrics for the quality assessments, namely:

Volume the larger a system, the more effort it takes to maintain since there is more information to be taken into account;
Redundancy duplicated code has to be maintained in all places where it occurs;
Unit size units as the lowest-level piece of functionality should be kept small to be focused and easier to understand;
Complexity simple systems are easier to comprehend and test than complex ones;
Unit interface size units with many parameters can be a symptom of bad encapsulation;
Coupling tightly coupled components are more resistant to change.

The first four properties have been introduced by Heitlager et al. (2007), whereas the last two have been recently added.

These indicators assess clearly defined aspects of maintainability. They can be calculated at least down to the unit level. This allows detailed analysis of the system when drilling down into the results later on.

2.2 Measurements aggregation

In order to support a reproducible evaluation, software quality measurement needs a clear mapping to an agreed standard. For that reason, the measurements are interpreted in the framework of a hierarchical quality model with dimensions according to the ISO/IEC 9126. In the ISO/IEC 9126 standard, maintainability is seen as a general quality characteristic of a software product and is decomposed into the subcharacteristics of analyzability, changeability, stability, and testability (International Organization for Standardization 2001).

In the model, the subcharacteristics are made quantifiable with the above source code metrics (Heitlager et al. 2007). For this, raw metrics have to be aggregated to the level of the whole system. This is done either by using a grand total (such as for Volume and Duplication) or by using so-called *quality profiles*. The latter summarize the distribution of a metric (e.g. cyclomatic complexity) at a certain level (e.g. per unit) by performing a classification into *risk categories* based on a set of thresholds. The outcome is percentage of code in low, moderate, high, and very high risk.

The aggregated measurements are used to determine a *rating* for each source code property, based on the application of another set of thresholds. These are further combined to calculate ratings for the subcharacteristics and the general maintainability score for a given system (Correia et al. 2009). The ratings correspond to 5 quality levels, represented by a number of stars, from ★ to ★★★★★. Details can be found in Appendix.

A validation study on open-source systems has shown that ratings as awarded by the quality model correlate positively with the speed with which defects are resolved by the system's maintainers (Luijten and Visser 2010). Also, the impact of technical quality on various external issue handling indicators was quantified, and the results corroborate the validity of the model (see Bijlsma 2010). For example, for systems rating 4 stars, issues were found to be resolved 3 times faster than for systems rating 2 stars.

As depicted in Fig. 1, the layered quality model provides the criteria for the evaluation procedure.

2.3 The role of a software benchmark repository

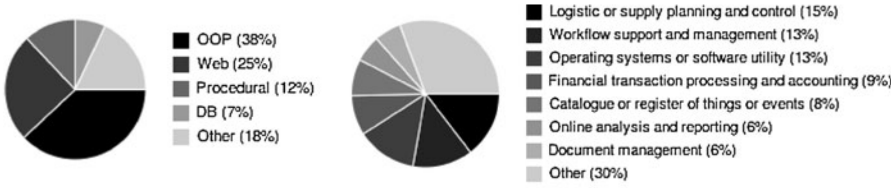
Even with quality dimensions derived from an international standard, quality indices calculated from source code measurements remain arbitrary as long as no comparison with other systems is available. To provide such information, SIG maintains a benchmark repository (Correia and Visser 2008a) holding the results from several hundreds of standard system evaluations carried out so far. As shown in Fig. 1, the benchmark database is updated with the results of each evaluation that is carried out.

Currently (May 2010), the benchmark repository holds results for over 500 evaluations (measurements computed for a language within a system) encompassing around 200 systems. These comprise proprietary systems (about 85%) as well as open-source systems. A total of 45 different computer languages are used by the systems in the repository, with Java, C, COBOL, C#, C++, and ABAP as largest contributors in terms of lines of code.

The repository entries are described with meta-data to characterize the systems along several dimensions. Information like functionality, development model, or architecture (among others), is added by the analysts involved in each project. In Table 1, average values for some key metrics are listed per represented programming paradigm. In Fig. 2,

Table 1 Average values for redundancy and complexity

Paradigm or group	Redundant lines (%)	Decision density (McCabe / LOC) (%)
OOP (e.g. Java, C#)	12.3	19.6
Web (e.g. JSP,ASP,PHP)	32.5	15.5
Procedural (e.g. C, COBOL)	20.2	10.0
DB (e.g. PL/SQL, T-SQL)	28.2	7.9



(a) Percentage of systems per paradigm.

(b) Percentage of systems per functionality group.

Fig. 2 Overview of the contents of the software benchmark repository per programming paradigm and functionality group. For the latter, we employ a taxonomy used by ISBSG Lokan (2008) we show some examples of the heterogeneity of the contents of this repository in terms of programming paradigm and functionality.

With this extensive statistical basis, SIG can compare any system to the whole repository or to similar products in terms of size, programming languages, or industry branch. Furthermore, interpreting the ranking, an evaluator can guide his scrutiny to parts of the code really needing improvement rather than curing minor symptoms. Such comparisons and interpretations are performed in the context of the quality consultancy services described in Sect. 4.2.

2.4 Calibration of the quality model

The evaluation data accumulated in the benchmark repository are also used for calibrating the quality model.

Calibration is performed on two different levels, namely to determine thresholds for (1) the raw metrics and for (2) the aggregated quality profiles. The first level of calibration is performed by analyzing the statistical distributions of the raw metrics among the different systems. Thresholds are then determined based on the variability between the systems, allowing us to pinpoint the more uncommon (thus considered riskier) range of values for the metric. More details on the methodology used for the first-level calibration were laid out by Alves et al. (2010).

For the second level, the aggregated quality profiles per system are used and thresholds for those are tuned in such a way that for each lowest-level source code property a desired symmetrical distribution of systems over quality ratings is achieved. Concretely, the model is calibrated such that systems have a $\langle 5, 30, 30, 30, 5 \rangle$ percentage-wise distribution over 5 levels of quality. This means that if a system is awarded 5 stars, it is comparable to the 5% best systems in the benchmark, in terms of maintainability. Note that it would be possible to select any other distribution, since it is a parameter of the calibration algorithm. We chose this one in particular so that only very good systems attain 5 stars, hence promoting excellence.

It would be possible to rely on expert opinions to define the thresholds, but calibration against a large set of real-world systems brings some advantages, namely:

(a) the process is more objective since it is based solely on data;
(b) it can be done almost automatically, thus allowing to easily update the model;
(c) it creates a realistic scale, since it is constructed to represent the full range of quality achieved by real-world systems.

For the calibration, a subset of the repository is selected according to certain criteria. Namely, only evaluations for modern programming languages that pertain to recently developed systems (in the past decade) are taken into account. This ensures that the quality model remains a reflection of the state of the art in software engineering. That subset is then manually inspected and purged of outliers to ensure the reliability of the obtained results. The high number of included systems and the heterogeneity in terms of domains, languages, architectures, owners, and/or producers (SIG's clients and their suppliers, as well as open-source communities) help to guard the representativeness of the calibration set.

Such calibration is performed with an updated set of systems at least once per year. The changes in the data set are, nevertheless, kept small enough not to cause abrupt changes after re-calibration.

3 Standardized evaluation procedure

A standard evaluation procedure has been defined in which the SIG quality model is applied to software products (Correia and Visser 2008b). The procedure consists of several steps, starting with the take-in of the source code by secure transmission to the evaluation laboratory and ending with the delivery of an evaluation report (see Fig. 3).

Intake The source code is received via a secure upload and copied to a standard location. A checksum is calculated to allow for future identification of the original source.

Scope In this step, the scope of the evaluation is determined. As a result, an unambiguous description of which software artifacts are to be covered by the evaluation becomes available. This scope description includes information such as: the identification of the software system (name, version, etc.), a characterization of the technology footprint of the system (which programming languages and the number of files analyzed for each), as well as a description of specific files excluded from the scope of the evaluation and why.

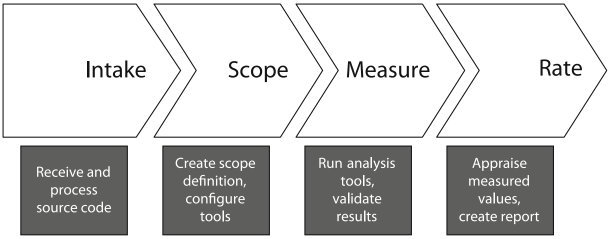
Measure In this step, a range of measurement values is determined for the software artifacts that fall within the evaluation scope. Each measurement is determined automatically by processing the software artifacts with an appropriate algorithm. This results in a large collection of measurement values at the level of source code units, which are then aggregated to the level of properties of the system as a whole, as described in Appendix.

Rate Finally, the values obtained in the measurement step are combined and subsequently compared against target values in order to determine quality subratings and the final rating for the system under evaluation.

The procedure conforms to the guidelines of the ISO/IEC 14598 standard for software product evaluation (International Organization for Standardization 1999), which is a companion standard to the ISO/IEC 9126. This is illustrated in Fig. 4.

To further ensure the objectivity and traceability of the evaluation, the evaluation laboratory of the SIG that carries out the procedure conforms to the guidelines of the ISO/IEC 17025 international standard for evaluation laboratories (International Organization for Standardization 2005). Among other things, this standard requires to have a quality management system in place that strictly separates the role of *evaluator* (who operates source code analysis tools and applies the quality model to produce an evaluation report) and the role of *quality officer* (who performs independent quality control on the activities and results of the evaluator).

Fig. 3 Evaluation procedure (image adapted from Correia and Visser 2008b)



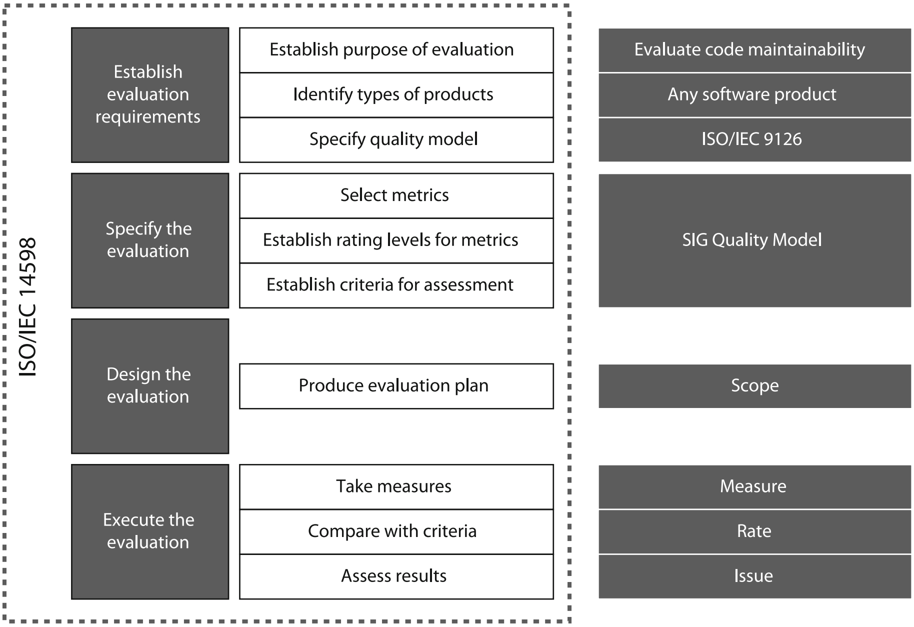


Fig. 4 Conformance to the ISO/IEC 14598 standard (image adapted from Correia and Visser 2008b)

4 Applications

The standardized procedure for measuring the maintainability of source code is used both in evaluation projects leading to certification and in consultancy projects leading to (management-level) recommendations.

4.1 Certification of maintainability

The availability of a benchmark repository provides the means for an objective comparison of software systems in terms of their maintainability. It is thus possible to assign an objective measure of maintainability to every system that undergoes the standardized evaluation procedure. This measure reflects the relative status of the system within the population of all systems evaluated so far.

Based on this system rating, TÜViT has set up a certification scheme. In this scheme, systems with maintainability scores above a certain threshold are eligible for the certificate called ‘TÜViT Trusted Product Maintainability’ (Software Improvement Group (SIG) and TÜV Informationstechnik GmbH (TÜViT) 2009) (see Fig. 5). To achieve a certificate, a system must score at least with 2 stars (★★) on all subcharacteristics and at least 3 stars (★★★) on the overall maintainability score. Besides reaching these minimal rankings, a system description is required to document at least the top-level components. The SIG software laboratory was accredited by the TÜViT certification body to function within this scheme as an evaluation laboratory for software code quality according to ISO/IEC Guide 65 (International Organization for Standardization 1996).

Fig. 5 The quality mark *Trusted Product—Maintainability*

As indicated in Fig. 1, the issued certificates are published in an online registry.[1] For full traceability, the certificates and the underlying evaluation reports are always annotated with the version of the quality model and source code analysis tools that were used for evaluation.

4.2 Standardized consulting approach

Based on experience from its assessment and monitoring projects in code quality and maintainability (van Deursen and Kuipers 2003; Kuipers and Visser 2004), SIG has enhanced the evaluation procedure described in Sect. 3 with activities to collect complementary information about a system under evaluation (Bouwers et al. 2009).

Usually, the projects start, after an initial meeting, with the source code submission. Next, a technical session is held together with the development staff of the customer to find out how the code is organized, what decisions were taken during development and so on. During a second, strategic session, SIG collects information from the management, i.e. the reason for the evaluation, history of the system, future plans, etc. With a method of structured business scenarios for the future use of the system, together with the customer, SIG attaches risk figures to the various development options the management has for the system under test. This information can be used later on to prioritize investments in code quality linking the quality assessment and the business plan for the system.

In parallel with these sessions, the standardized evaluation procedure is carried out. The results of the evaluation are communicated with the customers in several ways. To begin with, a validation session is scheduled to resolve results that may contradict the initial briefings. When the results are consolidated, SIG presents its findings in a management session. Finally, an assessment report is provided with the detailed results, and all recommendations for improvement established during the assessment.

4.3 Other applications

Standardized software evaluations have several other possible applications.

The evaluation of software maintainability provides managers in ongoing development projects with valuable information about the quality of the code they are producing, thus allowing direct *monitoring* of that quality (Kuipers and Visser 2004; Kuipers et al. 2007; Bouwers and Vis 2008).

Deciders *purchasing* software, in a scenario where the future maintenance burden will be on them, will reach better decisions when the code quality is evaluated for the products on their shortlist.

---

[1] http://www.tuvit.de/english/Maintainability.asp.

In *tendering* or *outsourcing* procedures, software providers may prove the quality of their product with a certificate.

Reaching a certifiable level of maintainability may become part of *development contracts* and thus raise the quality level of individual software projects.

5 Case examples

In this section, we recount some cases where the standardized approach described in this paper played a central role in the improvement of software maintainability.

5.1 Ministry of Justice, NRW, Germany

For its prison regime, the Ministry of Justice in the German state of Nordrhein-Westfalen uses BASIS-Web, a complete prison management system implemented in Java client–server technology. Since 2005, BASIS-Web computers are used for the management of prisoners data, the correct calculation of periods of detention as well as the treatment of cash balances of prisoners.

SIG has been asked, together with TÜViT, to assess the future proofness of this system. The assessment had two goals as follows: first to give insight into the maintainability of the system, secondly to determine whether any future performance risks can be identified from the source code. The system analysis was performed according to the standardized consulting approach described in Sect. 4.2.

The main result was insight into the overall maintainability of the system, based on the ISO/IEC 9126 dimensions of analyzability, changeability, testability, and stability. Additionally, the system was compared against a benchmark of other systems with similar size and architecture. This provided the Ministry of Justice as well as the development contractor insights into the strengths and risks.

The overall technical quality of the system was found to be somewhat lower than expected by the Ministry and the contractor and also somewhat lower than the average of comparable systems. Although this result may be explainable from the project's basic parameters (e.g. design phase started before standardized Java components for session control became available and thus had to be developed by the project), it clearly opens the stage for improvement because it replaced a number of subjective expert opinions about code quality with an objective assessment.

More in-depth analysis of the results on component and partly on unit level revealed particular places in the code to concentrate further effort on. SIG and TÜViT consultants gave a number of concrete measures for improvements in BASIS-Web. Every suggested improvement was validated in the repository for relevance in current state of the art Java software systems. This allowed to prioritize the improvements in a roadmap and to avoid costly cosmetic work in the source code without effect on later maintainability.

To help the customer relate possible improvement work and overall business strategy for the system, three different scenarios were considered as follows:

1. Continue using the system as is, i.e. with the current user base and functionality. However, even without enhancements in users or functions, a baseline of maintenance effort resulting from bug fixing or adaptation to changing legislation has to be expected.

Table 2 Quantification of savings due to monitoring (reproduced from van Hooren 2009)

	Developed with monitoring	Estimated rebuild value in man-months	Hours spent on defects	Number of defects
System A	Yes	34	<20	2
System B	No	89	500	25

The two systems are of comparable functional size and were developed and maintained within the same KAS BANK development team

2. Extend the user base for the system to more German states or to foreign countries. This scenario would mean more maintenance effort beyond the baseline from scenario 1 since more users demand bug fixes, updates, and enhancements. Additionally, the system performance has to be improved.
3. Cover more areas of prison management than today, thus enhance functionality. For this scenario, SIG and TÜViT foresaw the same challenge in maintenance terms as in scenario 2, this time being more functionality the driver for increase in code size and complexity.

For the first scenario (maintaining the status quo), only quick-wins and easy-to-implement improvements from the roadmap were suggested (e.g. fixing empty Java exceptions). For scenario 2 (extension of the user base) and 3 (extension of the functionality) however, major and costly reworkings of the code base have to be considered in order to yield a future-proof prison management system (e.g. partly change function allocation to modules, use a different paradigm for session control).

5.2 KAS BANK

KAS BANK is a European specialist in wholesale security services that had to meet the challenge of updating its IT systems. These systems had gradually become legacy, with two undesirable consequences, namely i) they posed a risk when new services were introduced and ii) they were costly with regard to maintenance (van Hooren 2009). KAS BANK decided to gradually modernize this application portfolio by migration to the .NET platform, making optimal use of its existing employees and their know-how.

KAS BANK decided to make use of SIG's portfolio monitoring service to safeguard the maintainability of the new software in the long run. In addition, the maintainability of the modernized software was certified by TÜViT. For example, the *Tri-Party Collateral Management* system of KAS BANK achieved a certificate with 4 stars in April 2009.

KAS BANK analyzed the savings that were obtained through monitoring the systems during their development in terms of maintenance effort and numbers of defects occurring in the post-production stage. A summary is given in Table 2. Although these results are not based on a controlled experiment, they give an indication of the positive effect of monitoring in both the reduction of the number of hours that needed to be spent on solving defects and in the reduction of the number of defects that occurred.

6 Discussion of the approach

In the following, we will discuss a number of potential limitations to our approach.

6.1 Repository bias

The benchmark repository that supports calibration consists of systems analyzed by SIG. This could lead to a bias in the repository, which would have an impact on the generalizability of the quality model. An important type of bias could be that only systems with quality problems require SIG's services, thus resulting in low standards for quality.

We think that a systematical bias of this kind is unlikely, since the repository is populated with systems analyzed in different contexts, for which varying levels of quality are to be expected. It contains systems analyzed in the context of (1) one-off assessments, (2) monitoring projects, (3) certification requests, and (4) assessments of open-source. Systems analyzed for (1) may actually have quality problems, those analyzed for (2) typically are steered to improve their quality. In the context of (3), good quality systems are to be expected and (4) are performed by SIG's own initiative. Thus, systems analyzed in the context of (2), (3), and (4) should display state of the art code quality, whereas only systems analyzed in the context of (1) might suffer from quality problems leading to the involvement of SIG for improvement.

Every time calibration is performed, we conduct a systematic investigation of the bias in the set of systems used. This is done using statistical analysis to compare the ratings obtained by different groups of systems. Unfortunately, it is virtually impossible to investigate all potential dimensions of bias. Currently, we inspect bias with respect to volume (is there a significant difference between big and small systems?), programming languages (e.g. do Java systems score differently from C# systems?), SIG project type (one-off assessments, monitoring projects, certification requests and assessments of open-source), development environment (industrial development versus open-source), among others.

6.2 Quality model soundness

An important part of the approach presented in this paper is the quality model used to aggregate source code metrics to ratings. This model was created to be pragmatic, easy to explain, technology independent and to enable root-cause analysis (see Heitlager et al. 2007). As any model, it is not complete and provides only an estimation of the modeled variable, in this case maintainability.

Even though the model was created and developed through years of experience in assessing maintainability, by experts in the field, it is important to scientifically assess its soundness. We have been conducting empirical studies in order to build up more confidence in the model and its relationship with actual maintainability of a system.

Correia et al. (2009) conducted a study on the connections between source code properties and ISO/IEC 9126 subcharacteristics, as defined in the model. The relationship was found to be mostly consistent with the expert opinions. Luijten and Visser (2010) performed another study, this time to investigate the relationship between the ratings calculated by our quality model, and the time taken to solve defects. All ratings, except for one, were found to correlate with defect resolution time, which is a proxy for actual maintenance performed. This study was further extended in a Master's project (Bijlsma 2010) to include enhancements, 3 more indicators of issue handling performance, and quantification of the relationship. The results were consistent with (Luijten and Visser 2010) in supporting the relationship between issue handling performance and ratings.

All three studies further developed our confidence in the model, but also helped reveal some limitations. We continue to extend these studies and perform new ones, as well as progressively improving the model.

6.3 Quality model stability

As described in Sect. 2.4, re-calibration is performed periodically, at least once a year. The objective is to ensure that the quality model reflects the state of the art in software engineering.

An important issue stemming from re-calibration could be that updating the set of systems would cause the model to change dramatically, thus reducing its reliability as a standard. As mentioned in Sect. 2.4, this is taken into consideration when determining the modifications (addition of new systems, removal of old ones) to the set.

A recent (January 2010) re-calibration was evaluated for its impact on 25 monitored systems. This was performed by calculating, for each system, the differences in ratings obtained by applying the existing model, versus the ones obtained by applying the newly calibrated one. The result was an average difference of 0.17 in terms of the overall *maintainability* rating per system, ranging from -0.56 to $+0.09$. Ratings are calculated in a continuous scale in the range $[0.5, 5.5[$ thus these values correspond to 3.4, -11.2 and $+1.8\%$ of the possible range (respectively). We think that such a small change per re-calibration is what to expect from the repository in the view of real improvements in software quality.

It could be argued that re-calibration weakens the role of the quality model as a standard for software quality because there is no clear reference to the particular parameter set used for an evaluation. To counter this, the quality model is explicitly versioned and any document related to evaluation results identifies the version of the model used. If there is doubt that a particular current quality model is correct, evaluation can also be done with the parameter set of an earlier model and results can be compared.

6.4 Focus

The standardized evaluation procedure described in this paper has its focus on assessing a software product's maintainability. As described in the ISO/IEC 9126 (International Organization for Standardization 2001), this is just one aspect of a software product's quality, thus it is possible and even desirable to use it in combination with other quality instruments.

Various kinds of *software testing*, such as unit testing, functional testing, integration testing, and acceptance testing, are essential for software product quality, both functional and technical. However, evaluation of technical quality as in the described approach does not replace testing, but operates on a higher level, is less labor-intensive, and can be performed independently.

Methodologies for *software process improvement* (SPI), such as the Capability Maturity Model Integration (CMMI), concern the production process, rather than the product. SPI works under the assumption that better processes lead to better products. Since this relationship is not a perfect one, improving software via an objective assessment of source code quality is an independent approach usable in a complementary way.

7 Related work

7.1 Software product quality improvement

The idea of improving software quality with the use of source code metrics has a long history. An interesting recent work is the Code Quality Management (CQM) framework proposed by Simon et al. (2006). Besides applying code metrics in large software projects, the authors introduce the idea of a benchmark repository for comparing between projects and identifying the best practices across the software industry. However, current emphasis in CQM is given to the definition of new creative quality indicators for object-oriented programming rather than to setting up a universal benchmark standard for comparison across the different software development paradigms.

7.2 Quality assessment methodologies

Some methodologies have been proposed for the assessment of software products, namely targeted at open-source project. These include OSMM Golden (2005), QSOS Atos Origin (2006), and OpenBRR OpenBRR.org (2005). A comparison of the latter two can be found in Deprez and Alexandre (2008). These methods mainly focus on the community contribution, activity, and other 'environmental' characteristics of the product. Although these approaches usually include assessment of technical quality, no concrete definition of measurements or norms are provided.

Currently, there are several research projects related to quality of open-source software, e.g. FLOSSMetrics, QualOSS, or SQO-OSS.[2] Each of these three projects aims to provide a platform for gathering information regarding open-source projects and possibly to provide some automated analysis of quality.

7.3 Certification of functional quality

Here, we briefly discuss work related to the application of our standardized evaluation procedure for software product certification.

Heck and Eekelen (2008) have developed a method for software certification where five levels of verification are distinguished. At the highest level, the software product is verified using formal methods where not only properties are proven about an abstract model, but about the software itself.

ISO/IEC 9126 lists security as one of the subcharacteristics of functionality. The most well-known software standard regarding security is the ISO/IEC 15408 standard on evaluation of IT security criteria International Organization for Standardization (2005) It is also published under the title *Common Criteria for Information Technology Security Evaluation* (CC) and is the basis for an elaborate software product certification scheme. Besides specific product security requirements, this standard defines a framework for specification, implementation, and evaluation of security aspects. In many industrial countries, there are national certification schemes for software product security based on the CC.

The ISO/IEC 25051 (International Organization for Standardization 2006) specifies requirements for functionality, documentation, and usability of Commercial Off-The-Shelf (COTS) products. COTS products are standard software packages sold to consumers

---

[2] http://flossmetrics.org, http://www.qualoss.eu, http://www.sqo-oss.org.

‘as is’, i.e. without a consulting service or other support. ISO/IEC 25051 requires that any claim made in the product documentation is tested, thus assuring the functional correctness of the product. The fulfillment of this standard can also be certified by many certification bodies.

Finally, the international standard ISO 9241 (International Organization for Standardization 2008) describes requirements for software product usability. ISO 9241 pursues the concept of usability-in-context, i.e. usability is not a generic property but must always be evaluated in the context of use of the product. In Germany, there is a certification scheme for software product usability based on ISO 9241-110, -11.

7.4 Software process quality improvement

The issue of software quality can be addressed not only from the point of view of the product, but also by evaluating and improving the development process. Originating from the so-called software crisis and the high demands on software quality in the defense sector, process approaches to software quality have been around for some twenty years.

Process approaches like Capability Maturity Model Integration[3] (CMMI) or Software Process Improvement and Capability Determination (SPICE) (International Organization for Standardization 2004) can best be seen as collections of best practices for organizations aiming at the development of excellent software. Both approaches arrange best practices in reference process models.

A basic concept is the maturity of an organization in implementing the quality practices and the reference models. The maturity concept motivates to improve once an initial level is reached. In several industrial areas, a minimal maturity level is required for contractors to succeed in software development business. Thus, together with the models, audit and certification schemes are available to assess the process maturity of an organization. Training institutions provide insight into the models and support candidate organizations in improving their maturity level.

Although providing some probability, process approaches to software quality cannot guarantee that a software product reaches a certain technical quality level. This is because the impact of the process on the actual programming work is only indirect. Additionally, implementing process approaches usually requires some investments since a number of roles have to be staffed with trained personnel which gets removed from the creative development process.

7.5 Benchmarking

Software benchmarking is usually associated with productivity rather than code quality. Jones (2000) provides a treatment of benchmarking software projects. The focus is not on the software product, though the functional size of systems in terms of function points and the technical volume in terms of lines of code are taken into account.

The International Software Benchmarking Standards Group (ISBSG)[4] collects data about software productivity and disseminates the collected data for benchmarking purposes. Apart from function points and lines of code, no software product measures are taken into account.

---

[3] See http://www.sei.cmu.edu/cmmi/.

[4] See http://www.isbsg.org.

Izquierdo-Cortazar et al. (2010) use a set of 1,400 open-source projects to determine thresholds for a number of metrics regarding the level of activity of communities. This is comparable to how we calibrate our quality model, but their work differs in terms of focus (community quality), repository composition (restricted to open-source), and methodology.

8 Concluding remarks and future work

We have provided an overview of the standardized models and procedures used by SIG and TÜViT for evaluation, benchmarking, and certification of software products. Standardization is achieved by following terminology and requirements of several relevant international standards.

We have explained the role of a central benchmark repository in which evaluation results are accumulated to be used in annual calibration of the measurement model. Such calibration enables comparison of software products against industry-wide levels of quality. In combination with standardized procedures for evaluation, the calibrated model is a stable basis for the presented software product certification scheme.

We have shown, with real-world examples, the value of the approach as a tool for improving and managing technical quality.

We are continuously working on evaluating and validating the quality model in various ways (see Correia et al. 2009; Luijten and Visser 2010). We also plan to extend the model to encompass more dimensions of technical quality, such as implemented architecture (Bouwers et al. 2009), or even other dimensions of software quality besides maintainability.

Furthermore, given the importance of the software benchmark repository and its size and heterogeneity, we aim to continuously extend it with new and more systems.

Appendix: The quality model

The SIG has developed a layered model for measuring and rating the technical quality of a software system in terms of the quality characteristics of ISO/IEC 9126 (Heitlager et al. 2007). The layered structure of the model is illustrated in Fig. 6.This appendix section describes the current state of the quality model, which has been improved and further operationalized since (Heitlager et al. 2007).

Source code metrics are used to collect facts about a software system. The measured values are combined and aggregated to provide information on properties at the level of the entire system, which are then mapped into higher level ratings that directly relate to the

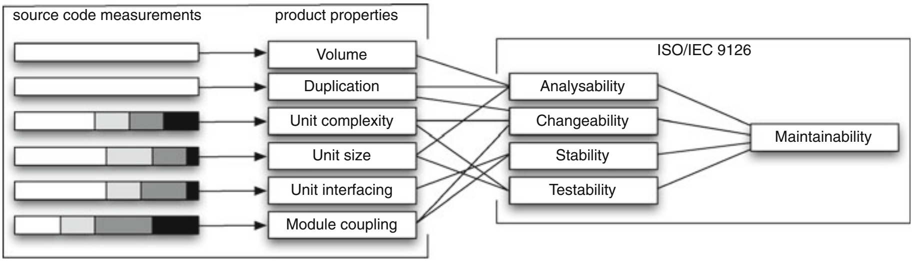


Fig. 6 Relation between source code metrics and system subcharacteristics of maintainability (image taken from Luijten and Visser 2010)

ISO/IEC 9126 standard. These ratings are presented using a five star system (from ★ to ★★★★★), where more stars mean better quality.

Source code measurements

In order to make the product properties measurable, the following metrics are calculated:

Estimated rebuild value The software product's rebuild value is estimated from the number of lines of code. This value is calculated in man-years using the Programming Languages Table of the Software Productivity Research Software Productivity Research (2007). This metric is used to evaluate the *volume* property;

Percentage of redundant code A line of code is considered redundant if it is part of a code fragment (larger than 6 lines of code) that is repeated literally (modulo white-space) in at least one other location in the source code. The percentage of redundant lines of code is used to evaluate the *duplication* property;

Lines of code per unit The number of lines of code in each unit. The notion of unit is defined as the smallest piece of invokable code, excluding labels (for example a function or procedure). This metric is used to evaluate the *unit size* property;

Cyclomatic complexity per unit The cyclomatic complexity (McCabe 1976) for each unit. This metric is used to evaluate the *unit complexity* property;

Number of parameters per unit The number of parameters declared in the interface of each unit. This metric is used to evaluate the *unit interfacing* property;

Number of incoming calls per module The number of incoming invocations for each module. The notion of module is defined as a delimited group of units (for example a class or file). This metric is used to evaluate the *module coupling* property.

From source code measurements to source code property ratings

To evaluate measurements at the source code level as property ratings at the system level, we make use of just a few simple techniques. In case the metric is more relevant as a single value for the whole system, we use thresholding to calculate the rating. For example, for duplication we use the amount of duplicated code in the system, as a percentage, and perform thresholding according to the following values:

Rating	Duplication
★★★★★	3%
★★★★	5%
★★★	10%
★★	20%
★	–

The interpretation of this table is that the values on the right are the maximum values the metric can have that still warrant the rating on the left. Thus, to be rated as ★★★★★ a system can have no more than 3% duplication, and so forth.

In case the metric is more relevant at the unit level, we make use of so-called *quality profiles*. As an example, let us take a look at how the rating for unit complexity is calculated. First the cyclomatic complexity index (McCabe 1976) is calculated for each code unit (where a unit is the smallest piece of code that can be executed and tested individually, for example a Java method or a C function). The values for individual units are then aggregated into four *risk categories* (following a similar categorization of the Software Engineering Institute), as indicated in the following table:

Cyclomatic complexity	Risk category
1–10	Low risk
11–20	Moderate risk
21–50	High risk
>50	Very high risk

For each category, the relative volumes are computed by summing the lines of code of the units that fit in that category, and dividing by the total lines of code in all units. These percentages are finally rated using a set of thresholds, defined as in the following example:

Rating	Maximum relative volume		
	Moderate	High	Very high
★★★★★	25%	0%	0%
★★★★	30%	5%	0%
★★★	40%	10%	0%
★★	50%	15%	5%
★	–	–	–

Note that this rating scheme is designed to progressively give more importance to categories with more risk. The first category ('low risk') is not shown in the table since it is the complement of the sum of the other three, adding up to 100%. Other properties have similar evaluation schemes relying on different categorization and thresholds. The particular thresholds are calibrated per property, against a benchmark of systems.

Such quality profiles have as an advantage over other kinds of aggregation (such as summary statistics like mean or median value) that sufficient information is retained to make significant quality differences between systems detectable (see Alves et al. 2010) for a more detailed discussion).

The evaluation of source code properties is first done separately for each different programming language, and subsequently aggregated into a single property rating by weighted average, according to the relative volume of each programming language in the system.

The specific thresholds used are calculated and calibrated on a periodic basis based on a large set of software systems, as described in Section 2.4.

Continuous scale

The calculation of ratings from source code metrics is described in terms of discrete *quality levels*. These values will need to be further combined and aggregated and for that, a discrete scale is not adequate. We thus use the discrete scale for describing the evaluation schemes, but make use of interpolation to adapt them in order to obtain ratings in a continuous scale in the interval [0.5, 5.5[. An equivalence between the two scales is established so that the behavior as described in terms of the discrete scale is preserved.

Let us consider a correspondence of the discrete scale to a continuous one where ★ corresponds to 1, ★★ to 2 and so forth. Thresholding as it was described can then be seen as a step function, defined, for the example of duplication (d), as:

$$rating(d) = \begin{cases} 5 & \text{if } d \leq 3\% \\ 4 & \text{if } 3\% < d \leq 5\% \\ 3 & \text{if } 5\% < d \leq 10\% \\ 2 & \text{if } 10\% < d \leq 20\% \\ 1 & \text{if } d > 20\% \end{cases}$$

This step function can be converted into a continuous piecewise linear function as follows:

1. In order for the function to be continuous, the value for the point on the limit between two steps (say, for example, point 3% which is between the steps with values 4 and 5) should be between the two steps' values (in the case of point 3% it would then be $(4 + 5)/2 = 4.5$). Thus, for example, *rating* (5%) = 3.5 and *rating* (10%) = 2.5;
2. Values between limits are computed by linear interpolation using the limit values. For example, *rating* (5.1%) = 3.4 and *rating* (7.5%) = 3.

The equivalence to the discrete scale can be established by arithmetic, round half up rounding.

This approach has the advantage of providing more precise ratings. Namely, with the first approach we have, for example, $rating(5.1\%) = rating(10\%) = 3$, whereas in the second approach we have $rating(5.1\%) = 3.4 \approx 3$ and $rating(10\%) = 2.5 \approx 3$. Thus, one can distinguish a system with 5.1% duplication from another one with 10%, while still preserving the originally described behavior.

The technique is also applied to the evaluation schemes for quality profiles of a certain property. Namely, interpolation is performed per risk category, resulting in three provisional ratings of which the minimum is taken as the final rating for that property.

From source code property ratings to ISO/IEC 9126 ratings

Property ratings are mapped to ratings for ISO/IEC 9126 subcharacteristics of maintainability following dependencies summarized in a matrix (see Table 3). In this matrix, a × is placed whenever a property is deemed to have an important impact on a certain subcharacteristic. These impacts were decided upon by a group of experts and have further been studied in Correia et al. (2009).

The subcharacteristic rating is obtained by averaging the ratings of the properties where a × is present in the subcharacteristic's line in the matrix. For example, *changeability* is represented in the model as affected by *duplication*, *unit complexity* and *module coupling*, thus its rating will be computed by averaging the ratings obtained for those properties.

Table 3 Mapping of source code properties to ISO/IEC 9126 subcharacteristics

ISO 9126 maintainability	Properties					
	Volume	Duplication	Unit size	Unit complexity	Unit interfacing	Module coupling
Analyzability	×	×	×			
Changeability		×		×		×
Stability					×	×
Testability			×	×		

Finally, all subcharacteristic ratings are averaged to provide the overall *maintainability* rating.

References

Alves, T. L., Ypma, C., & Visser, J. (2010). Deriving metric thresholds from benchmark data. In *26th IEEE international conference on software maintenance (ICSM 2010)*, September 12–18, 2010, Timisoara, Romania.

Atos Origin. (2006). Method for qualification and selection of open source software (QSOS), version 1.6.

Bijlsma, D. (2010). *Indicators of issue handling efficiency and their relation to software maintainability*. Master's thesis, University of Amsterdam.

Bouwers, E., & Vis, R. (2008). Multidimensional software monitoring applied to erp. In C. Makris & J. Visser (Eds.), *Proceedings of 2nd international workshop on software quality and maintainability*. Elsevier, ENTCS, to appear.

Bouwers, E., Visser, J., & van Deursen, A. (2009). Criteria for the evaluation of implemented architectures. In *25th IEEE international conference on software maintenance (ICSM 2009)* (pp. 73–82). IEEE, Edmonton, Alberta, Canada, September 20–26, 2009.

Correia, J., & Visser, J. (2008a). Benchmarking technical quality of software products. In wcre '08: *Proceedings of the 2008 15th working conference on reverse engineering* (pp. 297–300). IEEE Computer Society, Washington, DC, USA, http://dx.doi.org/10.1109/WCRE.2008.1.

Correia, J. P., & Visser, J. (2008b). Certification of technical quality of software products. In L. Barbosa, P. Breuer, A. Cerone, & S. Pickin (Eds.), *International workshop on foundations and techniques bringing together free/libre open source software and formal methods (FLOSS-FM 2008) & 2nd international workshop on foundations and techniques for open source certification (OpenCert 2008)* (pp. 35–51). United Nations University, International Institute for Software Technology (UNU-IIST), Research Report 398.

Correia, J. P., Kanellopoulos, Y., & Visser, J. (2009). A survey-based study of the mapping of system properties to iso/iec 9126 maintainability characteristics. In *25th IEEE international conference on software maintenance (ICSM 2009)* (pp. 61–70), September 20–26, 2009. Edmonton, Alberta, Canada: IEEE.

Deprez, J. C., & Alexandre, S. (2008). Comparing assessment methodologies for free/open source software: OpenBRR and QSOS. In *PROFES*.

Golden, B. (2005). Making open source ready for the enterprise: The open source maturity model, whitepaper available from, http://www.navicasoft.com.

Heck, P., & van Eekelen, M. (2008). *The LaQuSo software product certification model: (LSPCM)*. Tech. Rep. 08-03, Tech. Univ. Eindhoven.

Heitlager, I., Kuipers, T., & Visser, J. (2007). A practical model for measuring maintainability. In *6th international conference on the quality of information and communications technology (QUATIC 2007)* (pp. 30–39). IEEE Computer Society.

International Organization for Standardization. (1996). ISO/IEC Guide 65: General requirements for bodies operating product certification systems.

International Organization for Standardization. (1999). ISO/IEC 14598-1: Information technology - software product evaluation - part 1: General overview.

International Organization for Standardization. (2001). ISO/IEC 9126-1: Software engineering—product quality—part 1: Quality model.

International Organization for Standardization. (2004). ISO/IEC 15504: Information technology—process assessment.

International Organization for Standardization. (2005a). ISO/IEC 15408: Information technology—security techniques—evaluation criteria for IT security.

International Organization for Standardization. (2005b). ISO/IEC 17025: General requirements for the competence of testing and calibration laboratories.

International Organization for Standardization. (2006). ISO/IEC 25051: Software engineering—software product quality requirements and evaluation (square)—requirements for quality of commercial off-the-shelf (cots) software product and instructions for testing.

International Organization for Standardization. (2008). ISO/IEC 9241: Ergonomics of human-system interaction.

Izquierdo-Cortazar, D., Gonzalez-Barahona, J. M., Robles, G., Deprez, J. C., & Auvray, V. (2010). FLOSS communities: Analyzing evolvability and robustness from an industrial perspective. In *Proceedings of the 6th international conference on open source systems (OSS 2010).*

Jones, C. (2000). *Software assessments, benchmarks, and best practices*. Reading: Addison-Wesley.

Kuipers T, & Visser, J. (2004a). A tool-based methodology for software portfolio monitoring. In M. Piattini, & M. Serrano (Eds.), *Proceedings of 1st international workshop on software audit and metrics, (SAM 2004)* (pp 118–128). INSTICC Press.

Kuipers, T., & Visser, J. (2004b) A tool-based methodology for software portfolio monitoring. In M. Piattini, et al. (Eds.), *Proceedings of 1st international workshop on software audit and metrics, (SAM 2004)* (pp. 118–128). INSTICC Press.

Kuipers, T., Visser, J., & de Vries, G. (2007). Monitoring the quality of outsourced software. In J. van Hillegersberg, J., et al. (Eds.), *Proceedings of international workshop on tools for managing globally distributed software development (TOMAG 2007)*, Center for Telematics and Information Technology, Netherlands.

Lokan, C. (2008). The Benchmark Release 10—project planning edition. Tech. rep., International Software Benchmarcking Standards Groups Ltd.

Luijten, B., & Visser, J. (2010). Faster defect resolution with higher technical quality of software. In *4th international workshop on software quality and maintainability (SQM 2010)*, March 15, 2010, Madrid, Spain.

McCabe, T. J. (1976). A complexity measure. In *ICSE '76: Proceedings of the 2nd international conference on software engineering* (p. 407). Los Alamitos, CA, USA: IEEE Computer Society Press.

OpenBRRorg. (2005). Business readiness rating for open source, request for comment. 1.

Simon, F., Seng, O., & Mohaupt, T. (2006). Code quality management: Technische Qualität industrieller Softwaresysteme transparent und vergleichbar gemacht. Heidelberg, Germany: Dpunkt-Verlag.

Software Improvement Group (SIG) and TÜV Informationstechnik GmbH (TÜViT). (2009). SIG/TÜViT evaluation criteria—Trusted Product Maintainability, version 1.0.

Software Productivity Research. (2007). Programming languages table (version 2007d).

van Deursen, A., & Kuipers, T. (2003). Source-based software risk assessment. In *ICSM '03: Proceedings of international conference on software maintenance* (p. 385). IEEE Computer Society.

van Hooren, M. (2009). KAS BANK and SIG - from legacy to software certified by TÜViT. Banking and Finance.

Author Biographies

Robert Baggen works as senior consultant for TÜViT since 1999. He has gained experience in software project and quality management, programming in major languages, usability and statistics. Recently, Robert involved in a number of product and process certification projects. He received scientific education in physics and psychology.

José Pedro Correia works as a researcher at the Software Improvement Group. In this role, he is involved in scientific research in the areas of software quality and source code analysis, as well as innovation projects within the company

Katrin Schill is educated as a computer scientist. She joined TÜViT in 2002. Katrin is experienced in code analysis, project and quality management as well as requirements engineering. In previous jobs she developed and coded software in the areas of data mining and environmental information systems.

Joost Visser is Head of Research & Development at the Software Improvement Group. In this role, Joost is responsible for innovation of tools and services, academic relations, and general research on software quality and metrics. Joost carried out his PhD work at the Dutch Center for Mathematics and Informatics (CWI) and has published over 40 articles in international conferences and journals.