TUM

INSTITUT FÜR INFORMATIK

Tagungsband 2. Workshop zur Software-Qualitätsmodellierung und -bewertung

Stefan Wagner, Manfred Broy, Florian Deissenboeck, Peter Liggesmeyer, Jürgen Münch (Hrsg.)

TUM-I0917
Juli 09

TECHNISCHE UNIVERSITÄT MÜNCHEN

Vorwort

Qualität ist seit Beginn der kommerziellen Entwicklung von Software ein wichtiges Thema in Forschung und Praxis und diese Bedeutung verstärkt sich noch weiter. Heutige Entwicklungen stellen zusätzliche Anforderungen an verschiedenste Qualitätsaspekte dar. Beispielsweise führt die Durchdringung von kritischen Systemen, wie Flugzeugen oder Automobilen, zu immer höheren Sicherheitsanforderungen an Software. Der starke Anstieg der durchschnittlichen Code-Größen und die Langlebigkeit von Software-Systemen machen die Wartbarkeit zu einer wichtigen Eigenschaft. Die Beherrschung von Software-Qualität stellt somit ein wichtiges Ziel im Software Engineering dar. Diesem Ziel steht aber die Komplexität und Vielschichtigkeit von Qualität gegenüber.

Es existiert eine große Zahl an unterschiedlichen Sichten und eine entsprechende Vielzahl an Herangehensweisen zu diesem Thema. Für die praktische Anwendung in der Software-Entwicklung stehen aufgrund dieser Vielfalt überwiegend nur Insellösungen zur Verfügung, die keine ganzheitliche Behandlung des Themas ermöglichen. Beispielsweise sind trotz der engen Verbindung Bewertungen von Zuverlässigkeit und Nutzbarkeit typischerweise nicht integriert.

Ein verbreitetes Vorgehen zur Bewältigung dieser Probleme stellt die Verwendung von Qualitätsmodellen und daraus abgeleiteter bzw. damit in Beziehung gesetzter Bewertungen dar. Ein solches Vorgehen wird sowohl in der Forschung untersucht, als auch bereits in der Praxis angewendet. Es hat sich aber oft gezeigt, dass Standards, wie die ISO 9126, nicht direkt anwendbar sind und eigene Qualitätsmodelle für spezifische Situationen erstellt werden müssen. Dies resultiert in teilweise sehr unterschiedlichen Ansätzen zur Qualitätsmodellierung und -bewertung. Ziel dieses Workshops war es, wie bereits in der ersten Ausgabe dieses Workshops (SQMB '08), diese Ansätze vorzustellen und zu diskutieren. Dies konnte durch ein breites Spektrum an Beiträgen von sehr technisch-orientierten bis hin zu sehr managementorientierten Modellen für eingebettete Systeme wie auch Informationssysteme erreicht werden.

Organisation

Der Workshop SQMB ’09 wurde in Zusammenarbeit der Technische Universität München und des Fraunhofer IESE organisiert. Der Workshop fand im Zusammenhang mit der Konferenz SE 2009 statt.

Organisatoren

Stefan Wagner, Technische Universität München
Manfred Broy, Technische Universität München
Florian Deißenböck, Technische Universität München
Peter Liggesmeyer, Fraunhofer IESE
Jürgen Münch, Fraunhofer IESE

Programmkomitee

Ralf Ackermann, SAP
Klaus Beetz, Siemens
Thomas Beil, Daimler
Manfred Broy, TU München
Horst Degen-Hientz, KUGLER MAAG CIE
Florian Deißenböck, TU München
Reiner Dumke, Universität Magdeburg
Gregor Engels, Universität Paderborn
Max Fuchs, BMW
Jürgen Knoblach, BMW
Peter Kock, MAN Nutzfahrzeuge
Christian Körner, Siemens
Claus Lewerentz, TU Cottbus
Peter Liggesmeyer, Fraunhofer IESE
Oliver Mäckel, Siemens
Jürgen Münch, Fraunhofer IESE
Dietmar Pfahl, Simula Research Laboratory
Markus Pizka, itestra
Reinhold Plösch, JKU Linz
Ralf Reussner, Universität Karlsruhe
Wilhelm Schäfer, Universität Paderborn
Kurt Schneider, Universität Hannover
Andy Schürr, TU Darmstadt
Dirk Voelz, SAP
Stefan Wagner, TU München
Andreas Zeller, Universität des Saarlandes
Rolf Ziegler, SAP

Externe Gutachter

Oliver Sudmann

Messen von Software-Qualität bei der SAP: der SAP Quality Index

Günther Limböck

SAP AG

Abstract. Der Vortrag geht auf die Herausforderungen ein, die mit dem Messen von Software-Qualität verbunden sind. Es wird ein Projekt bei der SAP vorgestellt, welches zum Ziel hatte, ein System zum Messen von Softwarequalität einzuführen. Wesentliche Inhalte des Vortrages:

- den Prozess der Definition von Key Performance Indikatoren für Software Qualität
- Überblick über die definierten KPIs sowie deren Aggregation in Form des SAP-Quality-Index
- Die Herausforderungen beim Schaffen der notwenigen Infrastruktur und Tools
- Konsequenzen für Governance und Betrieb des SAP-Quality-Index

Der Vortrag schließt mit einem Rückblick auf die Lessons Learned.

Comprehensive Landscapes for Software-related Quality Models

Michael Kläs, Jens Heidrich, Jürgen Münch, Adam Trendowicz

Fraunhofer Institute for Experimental Software Engineering,
Fraunhofer-Platz 1, 67663 Kaiserslautern, Germany
{michael.klaes, juergen.muench, jens.heidrich, adam.trendowicz}@iese.fraunhofer.de

Abstract. Managing quality (such as service availability or process adherence) during the development, operation, and maintenance of software(-intensive) systems and services is a challenging task. Although many organizations need to define, control, measure, and improve various quality aspects of their development artifacts and processes, nearly no guidance is available on how to select, adapt, define, combine, use, and evolve quality models. Catalogs of quality models as well as selection and tailoring processes are widely missing. One essential reason for this tremendous lack of support is that software development is a highly context-dependent process. Therefore, quality models always need to be adaptable to the respective project goals and contexts. A first step towards better support for selecting and adapting quality models can be seen in a classification of existing quality models, especially with respect to their suitability for different purposes and contexts. Such a classification of quality models can be applied to provide an integrated overview of the variety of quality models. This article presents the idea of so called comprehensive quality model landscapes (CQMLs), which provide a classification scheme for quality models and help to get an overview of existing quality models and their relationships. The article describes the usage goals for such landscapes, presents a classification scheme, presents the initial concept of such landscapes, illustrates the concept with selected examples, and sketches open questions and future work.

Keywords: Software Quality, Quality Assurance, Project Management, Quality Management, Quality Standards, Quality Definition, Quality Measurement

1 Introduction

The multitude of software-related quality models available and the lack of guidance for identifying, evaluating, selecting, and adapting a set of appropriate models for a specific organization or project implies 1) a need for getting a structured classification and overview of available quality models, 2) a need for linking quality aspects to higher-level goals of a project or an organization and the respective context, and 3) a need for appropriate selection and customization processes. This article focuses on the need for getting a structured classification and overview of available quality models.

Currently, a variety of quality models exists, originating from the literature, company standards, official standards, or other sources (e.g., they might be implicitly defined in measurement systems, key performance indicators, or quality gates). Typically, quality models focus on product quality (e.g., [1], [2]), process quality (e.g., maturity models, process adherence, or performance models), resource quality (e.g., server availability model, qualification model), or project quality (e.g., milestone slippage model or project cost). Each of these models usually supports only a limited set of application purposes (like characterization [3], improvement [4], or prediction [5]). In many cases, it is not obvious for which usage purposes the models are suitable, in which contexts they can be applied (e.g., in which application domain), and how to customize them. In addition, it is not clear to what extent the models have already been evaluated. In case of the availability of empirical evidence, evaluation and dissemination are typically limited to a specific context and difficult to find in the literature.

In consequence, quality assurance managers, quality managers, and project planners have significant problems in identifying the appropriate set of quality models that is relevant for them. Furthermore, the lack of a uniform classification of quality models aggravates the communication regarding quality aspects.

The concept sketched in this article consists of a classification of quality models and the use of this classification in a so-called comprehensive quality model landscape (CQML). The idea of cartographing quality models in landscapes can be compared with so-called Enterprise IT landscapes that aim at improving the communication regarding networked IT systems in a company through graphical visualizations [6].

The article is structured as follows: Section 2 defines a set of usage scenarios for CQMLs, Section 3 sketches the classification scheme, Section 4 gives an overview of the landscape concept, and Section 5 provides conclusions, open questions, and directions for future work.

2 Usage Scenarios for Comprehensive QM Landscapes

In the context of software engineering, a number of potential decision-making processes may be effectively supported by QM landscapes. In their research, the authors aim at the following particular CQMLs usage scenarios:

S1 Improve the communication between involved people (such as quality assurance personnel, managers, project planners, developers, contractors) regarding quality models;

S2 Provide a comprehensive overview of (relevant) quality models;

S3 Support the identification of gaps (i.e., identify areas where quality models would be necessary but are currently missing);

S4 Support the identification and selection of relevant quality models in a goal oriented way;

S5 Support the adaptation of quality models to specific goals and contexts (this also requires, for instance, an adaptation process);

S6 Support the identification of relationships between quality models;

S7 Support the combination of quality models (this also requires, for instance, aggregation and composition models as well as a kind of unique format for describing quality models).

3 Classification Scheme for Quality Models

The authors see the following initial requirements for the classification and the landscape: Classification categories need to be (1) meaningful/minimal in the sense that they contribute to at least one usage scenario, (2) complete in the sense that all quality models can be categorized, and (3) orthogonal in the sense that the classification is as unambiguous as possible.

In order to systematize and evaluate quality models, we created a classification scheme including major dimensions based on the goal template provided by the well-established Goal-Question-Metric (GQM) paradigm [7]. The GQM goal template specifies five aspects that should be considered while defining goals of software measurement. We utilize the GQM goal template as follows:

- *Object* specifies what is being examined by a quality model. The major classes of objects include products, processes, resources, and projects.
- *Purpose* specifies the intent/motivation of quality modeling. The (initial) major purposes include (ordered by an organization's measuring capabilities):
  - Characterization - describing objects with respect to quality,
  - Understanding - explaining dependencies between (sub-)qualities of objects,
  - Evaluation - assessing the achievement of quality goals,
  - Prediction - estimating the expected value of quality,
  - Improvement - determining what needs to be done for improving quality (quantitatively).
- *Quality Focus* specifies the quality attribute being modeled. Example software-related qualities are reliability of products, maturity of processes, productivity of personnel, or cost of projects.
- *Viewpoint (Stakeholder)* specifies the perspective from which the quality attribute is modeled. Typically, the perspective refers to a stakeholder from whose viewpoint the quality attribute is perceived.
- *Context* specifies the environment in which the quality modeling takes place. The context characteristics should, in particular, cover aspects such as:
  - *Scope*, which specifies the comprehension of an organizational and process area covered by a quality model. An example organizational scope could be the whole organization, business unit, group, or project, whereas an example process scope could be process, activity, or task.
  - *Domain*, which specifies the domain(s) a quality model covers (and is intended for). Typical software application domains include: embedded software systems, management & information systems, and web application.

These characteristics serve as a basis for pre-selecting major groups of quality models. As, in practice, each group will probably contain many, largely heterogeneous, models, additional aspects need to be considered in order to select a narrower group of quality models that will fit particular demands and capabilities. It must be ensured that a model's *critical prerequisites* are fulfilled before it can be utilized. One essential aspect to consider are inputs required by a specific quality model. This includes *type* of data (e.g., objective-subjective, categorical-numerical, or certain-uncertain), *amount of data*, and *quality of data* (e.g., completeness). Further, motivated by industrial demands, we propose considering such aspects as empirical evidence and utilization support. *Empirical evidence* specifies the amount and quality of empirical studies proving that a quality model works in practice as expected. *Utilization support* mainly refers to the amount and quality of documentation for a quality model; it may also include the existence of tools supporting the utilization of a quality model.

Finally, dependent on the values of the major characteristics, further aspects may need to be considered. For example, given that a quality model does not perfectly fit particular needs and capabilities its flexibility might be an essential characteristic to consider. *Flexibility* here refers to the extent to which the model (1) is already predefined (i.e., if it represents a fixed-model or a define-your-model approach) and (2) can be adapted to the specific needs and capabilities. Moreover, dependent on the purpose of modeling, the model's *availability* might play an important role. In case of prediction purposes, the earlier a prediction model can be applied (can provide estimates), the better.

4 Landscaping Quality Models

As mentioned in Section 1, there exists a variety of different quality models for different application scenarios, and it is a challenging task to create a landscape of existing models. For example, one usage of CQMLs should be the selection of appropriate quality models in a goal-oriented way for measurement goals defined on the basis of GQM (S4). Depending on the specific application scenarios of an organization, different dimensions may be important for creating such a landscape. Furthermore, it can be helpful to restrict the quality models presented in a landscape by fixing one or more dimensions. For instance, one might consider product quality models only (by restricting the "object" dimension) or one might only consider models relevant in a certain context (e.g., by restricting the scope/domain). Quality models may consist of several sub-models (e.g., ISO 9126 has models for internal quality, external quality, and quality in use). Depending on the level of detail of a particular landscape, those sub-models may have to be classified separately.

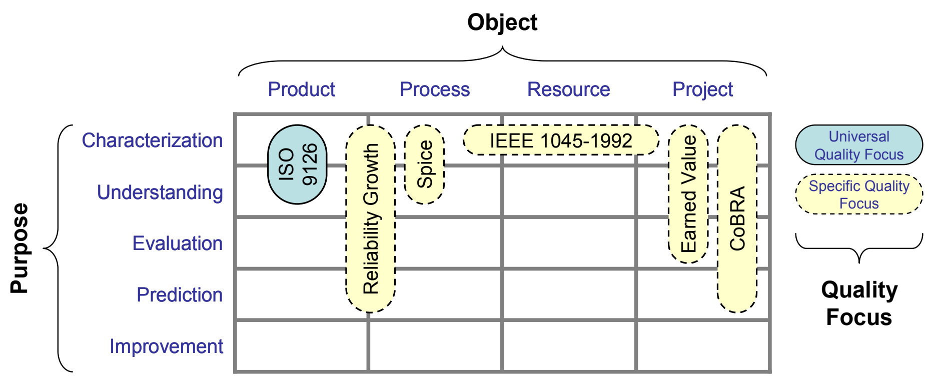


Fig. 1. Sample Landscape of Quality Models.

To give an example, let us consider three main dimensions for creating a high-level landscape of quality models: object, purpose, and quality focus. Fig. 1 gives a preliminary classification of some popular quality models according to those three dimensions. The landscape uses quite simple classes for the three dimensions:

- *Object:* A quality model may consider different objects. For instance, ISO 9126 [2] makes statements about the general software product quality, CoBRA [8] addresses project costs, and reliability growth models [5] make statements about actual product reliability or the remaining system test time of the test process. For our landscape, we want to classify the models into four classes: product, process, resource, and project. A quality model may be assigned to more than one class or even only address parts of a class (e.g., a certain sub-process like testing or a certain part of a product like the design document).
- *Purpose:* A quality model may support different purposes. In our landscape, we use the high-level classes: characterization, understanding, evaluation, prediction, and improvement. In practice, a quality model may support more specific purposes (like risk assessment or supplier management) that may be assigned to one or more of the high-level classes presented here. For instance, ISO 9126 may be used to characterize and understand software quality. There are no thresholds defined for the metrics in the model. So, evaluation (comparison) using ISO 9126 is not possible out-of-the-box. Furthermore, no prediction and quantitative improvement scenarios are supported.
- *Quality Focus:* Creating universal classes for the quality focus would probably be as difficult as creating a universal quality model. There exists a variety of qualities and sub-qualities, which are mostly ordered and defined differently in different quality models. For our landscape, we simply want to distinguish between quality models addressing a single, specific quality focus (like productivity as one aspect of process quality [3] or reliability as one aspect of product quality [5]) and models claiming to have a universal quality focus (usually refining qualities by sub-qualities), like the ISO 9126 for product quality.

5 Conclusions

This paper presents the first step towards developing a more comprehensive classification scheme for quality models and goal-oriented landscapes (so-called CQMLs). Future work will be in the area of refining and empirically evaluating the classification scheme for accessing and improving the applicability and usefulness for the stated scenarios. Existing quality models should be reviewed and classified based on the validated schema to build up a database of existing quality models. The classification scheme and landscapes should be integrated into a comprehensive model selection and adaptation process (inspired by the comprehensive reuse process [9]) for building up an experience base of quality models. This work is planned to be conducted as part of the publicly funded BMBF project “QuaMoCo”.

6 Acknowledgments

Parts of this work have been funded by the BMBF project “QuaMoCo – Software-Qualität: Flexible Modellierung und integriertes Controlling” (grant 01 IS 08 023 C).

7 References

1. Barry W. Boehm, John R. Brown, Hans Kaspar, Myron Lipow, Gordon J. MacLeod, and Michael J. Merritt, Characteristics of Software Quality. North Holland Publishing Company, 1978.
2. ISO/IEC 9126 International Standard, Software engineering – Product quality, Part 1: Quality model, 2001.
3. IEEE 1045 Standard for Software Productivity Metrics, 1992.
4. CMMI for Development, Version 1.2, CMU/SEI-2006-TR-008, Carnegie Mellon University, 2006.
5. Lyu M.R., Encyclopedia of Software Engineering. John Wiley & Sons, chapter Software Reliability Theory, 2002.
6. Matthes, F., Softwarekartographie. in: Kurbel, Karl; Becker, Jörg, Gronau, Norbert; Sinz, Elmar J.; Suhl, Leena (Hrs.): Enzyklopädie der Wirtschaftsinformatik, 1. Auflage, Oktober 2008.
7. Victor R. Basili: Software Modeling and Measurement: The Goal/Question/Metric paradigm, Technical Report CS-TR-2956, Department of Computer Science, University of Maryland, College Park, MD 20742, 1992.
8. L.C. Briand, K. El Emam, F. Bomarius, CoBRA: A Hybrid Method for Software Cost Estimation, Benchmarking and Risk Assessment, Proceedings of the 20th International Conference on Software Engineering, pp. 390-399, 1998.
9. V. R. Basili, H. D. Rombach, Support for comprehensive reuse, Software Engineering Journal, v.6 n.5, p.303-316, Sept. 1991.

A Proposal for a Quality Model Based on a Technical Topic Classification

R. Plösch[1], H. Gruber[1], C. Körner[2] , G. Pomberger[1], S. Schiffer[1]

(1) Johannes Kepler University Linz, Institute for Business Informatics – Software Engineering, Linz, Austria
reinhold.ploesch | harald.gruber | gustav.pomberger | stefan.schiffer @jku.at
(2) Siemens AG, Corporate Technology – SE 1, Otto-Hahn-Ring 6, Munich, Germany
christian.koerner@siemens.com

Abstract. Existing quality models like the ISO 9126 model lack preciseness in order to be able to assign metrics provided by static code analysis tools. Furthermore architects need a technical view on problems identified by static code analysis tools – independent of the classification a specific tool might provide. We therefore developed a Technical Topic Classification (TTC) that tries to overcome the problems mentioned above and assigned approximately 2,000 metrics from various static code analysis tools for Java, C#, and C++ to our TTC. The underlying metamodel for the TTC is semantically richer than the metamodel of typical quality models.

1 Introduction and Overview

According to IEEE 1061 [9] "*Software quality is the degree to which software possesses a desired combination of quality attributes*". Following this definition it is obvious that some mechanisms are necessary to systematically derive quality attributes. The general structure of such a model typically follows the Factor-Criteria-Metrics model (FCM) [1]. In this model the overall term quality is refined by quality attributes and criteria up to a detail level that makes it possible to formulate metrics. These metrics allow measuring the fulfillment of a piece of software with respect to a specific quality attribute. This is crucial, because otherwise it is not possible to find out whether a piece of software really possesses a desired quality (as requested by the IEEE 1061 definition). The GQM approach [17], [18], [19] uses this general FCM model and derives project-specific quality models by individually defining goals, questions and metrics. Other approaches try to formulate quality models that typically follow the FCM model and claim that they can be generally used in every software project. These general quality models can be tailored to the specific needs of the organization or project. Examples of such quality models are ISO 9126 [10], ISO 25000 [11], SATC [7] and FURPS [6] – just to name the most popular ones.

In our research project *Code Quality Management* (CQM) (see e.g. [13], [14] and [15]) the emphasis currently is on providing methods, models and tools for monitoring the quality of software throughout the development and maintenance phases of a project. One important requirement is to know the quality of the software (either generally or focused on some specific quality attributes) at any time with minimal effort. Having this as background we heavily apply static code analysis tools like PMD [16], FindBugs [5] or PC-Lint [12]. We therefore aim at operational quality models where the metrics provided by the static code analysis tools (examples are mentioned above) are assigned to the quality attributes. In our technical topic classification we currently have assigned more than 2,000 metrics to the various quality attributes. To build such a quality model it is first of all important for us that the model is defined in a way that the assignment of metrics to quality attributes can be done easily, by providing guidance for the assignment task in terms of rules and constraints. The quality model as defined by ISO 9126 (this is also valid for other published quality models) does not provide such guidance. It is for instance not easy to decide whether a metric like "Number of methods that do not adhere to the naming conventions" is an indicator for the quality attribute *Readability* or *Changeability* in the ISO 9126 quality model. It is simply not clear; the only guidance available is the definition of the quality attributes. In the specific case mentioned above the definition of the quality attributes *Readability* and *Changeability* do not give precise answers, though; see [3] for a more detailed discussion of shortcomings of the ISO 9126 quality model. Second, and even more important, we made the observation in various quality management projects that architects and technical project managers prefer a more technical view on quality. Although they are interested in maintainability (an ISO 9126 quality attribute) in general, they are more interested in a technical dimension – they want to find out more technically, why maintainability is bad. So they are interested in topics like exception handling, inheritance, declaration and definition, memory management, etc.

There are some approaches that try to provide a technical or problem classification (see e.g. [8], [4] and [2]). None of these approaches is comprehensive enough (a detailed analysis is outside the scope of this paper). From this background we developed a technical topic classification (TTC) that is presented in the subsequent chapters. Our principal approach is to provide a solid basis and a meaningful classification that especially supports our requirements that will be presented in Chapter 2. The TTC is extensible by nature, i.e. can be adapted to (domain) specific needs without corrupting the principle structure.

2 Technical Topic Classification

The ISO 9126 standard defines three types of quality – internal quality, external quality and quality in use. Our TTC focuses on internal quality, i.e. on the totality of characteristics of the software product from an internal view. Internal quality is measured and evaluated against the internal quality requirements. Details of internal quality can be improved during code implementation and reviewing [10]. We will show in the conclusion that the approach is flexible enough to cover external quality and quality in use, too. From the introduction it should be clear that the TTC has to adhere to a number of requirements:

- The classification scheme must address the typical topics for a technical project lead (or architect), i.e. runtime topics, design topics and code topics.
- The classification must be flexible enough to cope with different development paradigms, in particular with the imperative and the object-oriented paradigm.
- On each level of the quality model a clear guidance must be available that makes it easy to assign metrics to quality attributes.
- There are always situations where multiple assignments are necessary and useful. Nevertheless it must be made explicit in the quality model where this is appropriate and under which conditions.
- A metric must always be assigned to leaf elements in the quality model. If this is not forced by the quality model, shortcomings of the classification remain hidden.

Taking this as input a very general metamodel for our TTC can be given in Fig. 1. This metamodel is quite general and shows the principal frame of the TTC, but is of course not sufficient to understand the model itself. The top-level hierarchy of our TTC distinguishes *Runtime topics*, *Code topics* and *Design topics*. Runtime topics cover issues that are related to the execution of software. Design topics cover architectural issues and address abstract data types, classes, modules and systems, but explicitly not functions or expressions. Finally, code topics cover typical implementation related aspects like naming conventions, programming style, complexity, etc. Obviously the design topics could also be part of the code topics, but as design topics are of vital interest in a project we decided to give the design topics more weight by putting them on the top level of the TTC. The following guidance rules (constraints) apply for the assignment of metrics to the quality attributes:

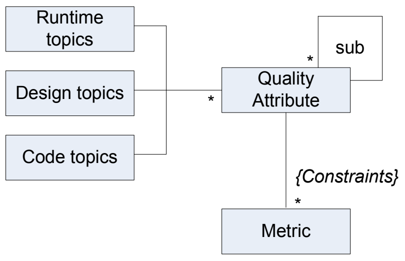


Fig. 1. Metamodel of the TTC

- A metric assigned to the runtime topics must be assigned to the code topics or the design topics, too.
- Not every metric has to be assigned to the runtime topics.
- Code topics and design topics are disjoint.

In the following subsections we will discuss the three main branches of the quality model in more detail.

2.1 Classification of Runtime Topics

Fig. 2 shows our classification of runtime topics. In this branch of the TTC multiple assignments of metrics to the top level quality attributes (*Concurrency*, *Distribution*, etc.) are possible. Within one of these quality attributes metrics have to be assigned uniquely to one leaf element of the sub-tree.

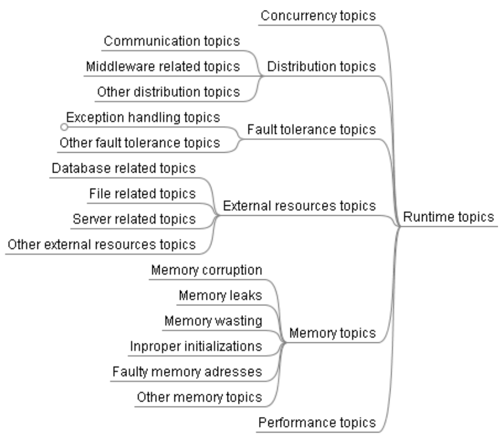


Fig. 2. Runtime Topics

On the top level of the classification we distinguish resource related topics (*Memory topics*, *Performance topics*, *External resources topics*), *Fault tolerance topics, Concurrency Topics* and *Distribution topics* (i.e. network related topics). On this abstract level all runtime related aspects can be assigned. For some topics (e.g. Concurrency topics, Performance topics) we do not yet provide a more detailed classification, as typical static code analysis tools do not provide a lot of metrics for these topics we also did not provide a more detailed classification here. On the most detailed level of the quality model there is always a leaf element named "*Other ... topics*". This is necessary for assuring, that metrics can always be assigned to a leaf element of the quality model. Furthermore these "Other … topic" elements are a good source for enhancing the TTC over time. It definitely makes sense to have a close look at metrics contained in this quality attribute and to find out whether there are some similarities that would justify changes to the quality model (renaming or refining of existing attributes, adding new attributes).

2.2 Classification of Code Topics

Fig. 3 shows the classification of code related topics. On the top level of this sub-hierarchy the classification of metrics is based on the programming language construct that a specific metric addresses. On this level five distinct categories are offered; the categories are named and defined in a way that imperative programming languages and object-oriented programming languages are both considered. Especially for the category *Expressions* it would have been possible to refine this into a more elaborate classification. We basically did not do this as architects are typically not interested in such details. If their focus is on code topics they might differentiate and focus on *Expressions* or *Functions*, but not on a specific kind of *Expression*.

The branches on this level are disjoint, so metrics must be assigned to one branch only. Unfortunately there is a practical exception to this rule. Some static code analyzers have badly designed rules. E.g., if they check naming conventions of methods and classes in one rule, a duplication of rules (i.e. multiple assignments) is inevitable.

On the next classification level we try to use the same classification topics for each abstract language construct wherever possible. This should make it considerably easier to assign metrics as the number of definitions that have to be kept in mind is smaller – without loosing preciseness of the classification.

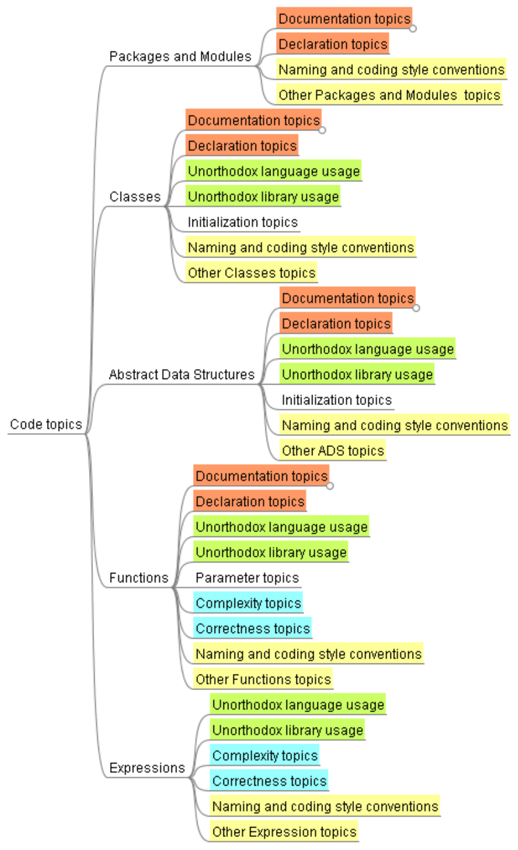


Fig. 3. Code Topics

There are different groups of metrics that are indicated by colors in Fig. 3 and have the following meaning:

Yellow: Applicable for all language constructs. Comprises the topics *Naming and coding style conventions* as well as the already discussed *Other … topics*.

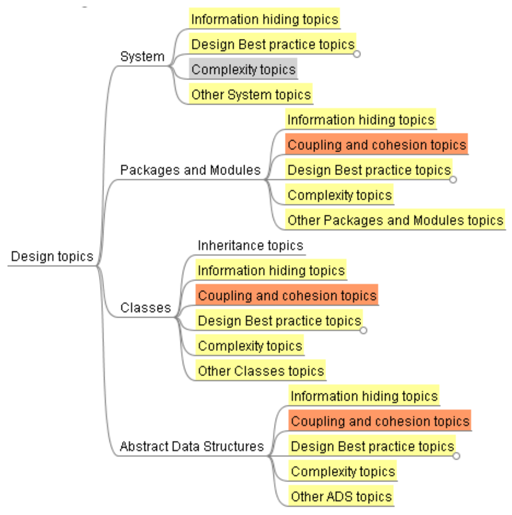
Red: Applicable for all categories except *Expressions*: This category comprises documentation and declaration topics. The documentation topics are refined (not shown in Fig. 3) in some sub-categories.

Green: Applicable for all categories except Packages and Modules. This category comprises unorthodox language usage as well as unorthodox library usage. This category might be refined later by providing additional attributes for different domains (i.e., unorthodox library usage for domain specific libraries) or subcategories formed by the programming language.

Cyan: Applicable for functions and expressions; comprises correctness and complexity topics. Besides these categorized topics (quality attributes) each topic can have distinct quality attributes, e.g. Parameter topics or initialization topics (see categories of *Functions* or *Classes* in Fig. 3). As a rule a metric may only be assigned to one category within the code topics. At this point of reading it seems strange that Complexity topics are assigned to Expressions and Functions, only. One would expect to have them on the class level as well as on the package and module level. In our opinion complexity beyond the expression and function layer has something to do with design; we therefore put Complexity for the higher-order language constructs in the design topics sub-branch (see next chapter).

2.3 Classification of Design Topics

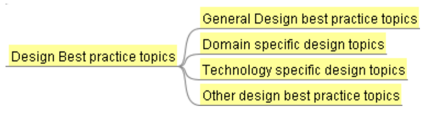
Fig. 4 shows the classification of design topics. On the top level of this sub-hierarchy the classification of metrics is based on typical design elements that a specific metric addresses. On this level four distinct categories are offered; they are named and defined in a way that both imperative programming languages and object-oriented programming languages are considered.



Design topics
System
Information hiding topics
Design Best practice topics
Complexity topics
Other System topics
Packages and Modules
Information hiding topics
Coupling and cohesion topics
Design Best practice topics
Complexity topics
Other Packages and Modules topics
Classes
Inheritance topics
Information hiding topics
Coupling and cohesion topics
Design Best practice topics
Complexity topics
Other Classes topics
Abstract Data Structures
Information hiding topics
Coupling and cohesion topics
Design Best practice topics
Complexity topics
Other ADS topics

Fig. 4. Design Topics

Again, on the next classification level we try to use the same classification topics for each abstract language construct wherever possible. This should make it considerably easier to assign metrics as the number of definitions that have to be kept in mind is smaller – without loosing preciseness of the classification. The two groups of metrics are indicated by different colors in Fig. 4:
Yellow: Applicable for all language constructs. Comprises the topics Information hiding topics, Design best practice topics, Complexity topics and the already known Other… topics. *Red:* Applicable for all categories except System. This category comprises coupling and cohesion topics. Besides these categorized topics (quality attributes) each topic can have distinct quality attributes – currently we have declared *Inheritance topics* for classes. One principal problem is that for example inheritance topics can be viewed as design best practice topics. The rule here is that identified practices (metrics) should first be assigned to the more specific topics before assigning it to the *Design best practice topics*. Again, the assignment of rules to these topics has to be disjoint. The design best practice topics are currently refined as shown in Fig. 5. These design best practice topics deal with commonly used design topics. The domain specific design topics can be refined (e.g. automation systems domain or embedded systems domain) as well as the technology specific design topics (e.g. CORBA topics, EJB topics, Message Queue topics or Transaction Management topics). Currently these refinements are not done.



Design Best practice topics
General Design best practice topics
Domain specific design topics
Technology specific design topics
Other design best practice topics

Fig. 5. Design best practice topics

3 Refined Metamodel of the TTC

With the information presented in section 2 of this paper we can reformulate our metamodel for the TTC and can therefore provide a model that gives more insight about the structuring mechanisms applied to the TTC – see Fig. 6.

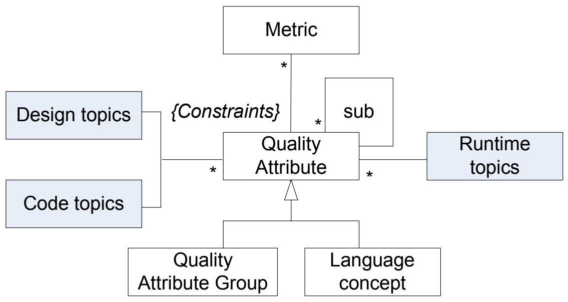


Fig. 6. Refined metamodel of the TTC

Language concept: The language concepts are to be understood as abstract language concept. In order to facilitate the meaning of the abstract concepts specific mappings of the abstract language concepts to the specific language concepts (e.g. for the programming language Java) could be provided. This would also ease the assignment process for metrics. *Quality attribute group:* This is a set of quality attributes that is used for more than one language concept. In the chapter 2 we marked these groups using a color code. *Other* … topics: This is a specific quality attribute that has to be contained as leaf topic for every quality attribute (for simplicity reasons this is not contained in Fig. 6). *Constraints:* On each level of the TTC it is explicitly defined by means of constraints to which extent multiple assignments of metrics to quality attributes are allowed. See chapter 2 for a more detailed description of the constraints and guidance rules.

4 Conclusion and Further Work

The quality model based on a technical classification is drawn on experience with technical classifications for quality management in a number of projects. A previous version of the TTC (unpublished) was used in more than 30 projects. The gained experience and the identified shortcomings led to this proposal of a technical classification.

Currently, only metrics from the tool PMD are assigned to the TTC – other tools will follow in the near future. This will possibly cause changes of the TTC; nevertheless we think that the metamodel will remain stable, as a number of possible enhancements and additions are already foreseen. *Domain specific design topics* currently are not refined into more specific topics – this will be done on demand, i.e. when tools arise that provide specific support for a specific domain.

The focus of the current TTC is on internal quality and on integration of metrics of static code analysis tools into the quality model. The integration of dynamic analysis tools (focusing on internal quality) should be possible without problems; the existing TTC is supposed to perfectly fit for that.

Adding quality attributes for external quality (e.g. correctness) will change the structure of the metamodel (as shown in Fig. 6) by adding e.g. *Correctness /*

Functionality on the top level of the TTC. Nevertheless other quality attributes relevant for external quality might need a deeper integration into the existing TTC (e.g. for runtime topics in the context of performance).

The refined metamodel as presented in this paper focuses on the structure of the quality model itself – other important aspects (besides the aspects already mentioned in this concluding chapter) of the model are not yet covered and have to be considered in the metamodel:

- *Justification of the Assignment of Metrics:* It is important to have a justification (which is structured information) for the assignment of metrics to quality attributes. This has to be modeled as Attribute on the relation between a metric and a quality attribute. Here some additional modeling is necessary that is not yet incorporated in the actual meta model.
- *Metrics details:* For an operational quality model detailed information about each metric (Name, description, tool, version of tool, example) has to be available, but also hints that help using these metrics (e.g., trustworthiness of the metrics, expected costs for fixing a problem related to this metrics, examples how to fix a problem)
- *Language concept details:* Besides a description of the abstract language concept details a mapping to programming language specific language concepts has to be provided.
- *Quality attribute details:* Besides a definition of the quality attribute, data about the completeness of the quality attribute has to be provided. From point of view of the metamodel this is quite easy as this can be realized by adding some additional data fields. Nevertheless it will be difficult to fill these data fields, as this needs a good understanding of what is missing. We think that this can be achieved more easily with our TTC (compared to classical quality models like ISO 9126), as it is easier to reason whether all *Naming Conventions* (one quality attribute of the TTC) are covered by tools, than reasoning, whether all aspects of *Readability* (a quality attribute of the ISO 9126 model) are covered.

Literature

1. Balzert H.: Lehrbuch der Software-Technik – Software Management; Software-Qualitätssicherung, Unternehmensmodelierung, Spektrum Akademischer Verlag, 1998
2. Beizer B.: Software Testing Techniques. Electrical Engineering, Computer Science and Engineering Series. Van Nostrand Reinhold, 1983.
3. Broy M., Deissenboeck F., Pizka M.: Demystifying maintainability. In: Proc. 4th Workshop on Software Quality (4-WoSQ). ACM Press, 2006.
4. Chillarege R., Bhandari I.S., Chaar J.K., Halliday M.J., Moebus D.S., Ray B.K., Wong M-Y.: Orthogonal defect classification – a concept for in-process measurements; IEEE Transactions on Software Engineering, Vol. 18, No. 11, November 1992, pp. 943-955
5. Findbugs: Product information about FindBugs can be obtained via http://findbugs.sourceforge.net
6. Grady R.B., Caswell D.L.: Software Metrics: Establishing a Company-Wide Program; (Prentice Hall, Upper Saddle River, NJ 1987)

7. Hyatt L., Rosenberg L.: A software quality model and metrics for identifying project risks and assessing software quality; In: Proceedings of 8th Annual Software Technology Conference, Utah, April 1996
8. IEEE: IEEE Standard Classification for Software Anomalies (IEEE 1044-1993), IEEE, 1993
9. IEEE Standard 1061-1998 Standard for a Software Quality Metrics Methodology, IEEE, 1998
10. DIN ISO 9126: Informationstechnik – Beurteilen von Softwareprodukten, Qualitätsmerkmale und Leitfaden zu deren Verwendung, 1991
11. ISO / IEC 25000:2005 – Software product Quality Requirements and Evaluation (SQuaRE)
12. Product information about PC-Lint can be obtained via http://www.gimpel.com/html/pcl.htm
13. Plösch R., Gruber H., Hentschel A., Körner Ch., Pomberger G., Schiffer S., Saft M., Storck S.: The EMISQ Method - Expert Based Evaluation of Internal Software Quality, Proceedings of 3rd IEEE Systems and Software Week, March 3-8, 2007, Baltimore, USA, IEEE Computer Society Press, 2007
14. Plösch R., Gruber H., Pomberger G., Saft M., Schiffer S.: Tool Support for Expert-Centred Code Assessments, Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation (ICST 2008), April 9-11, 2008, Lillehammer, Norwegen, IEEE Computer Society Press, 2008
15. Plösch R., Gruber H., Hentschel A., Körner Ch., Pomberger G., Schiffer S., Saft M., Storck S.: The EMISQ Method and its Tool Support - Expert Based Evaluation of Internal Software Quality, Journal of Innovations in Systems and Software Engineering, Springer London, Volume 4(1), March 2008
16. PMD: Product information about PMD can be obtained via http://pmd.sourceforge.net/
17. Rombach H.D., Basili V.R.: Quantitative Software-Qualitätssicherung; In: Informatik Spektrum, Band 10, Heft 3, Juni 1987, S 145-158
18. Solingen R.: The Goal/Question/Metric Approach; In: Encyclopedia of software Engineering, two-volume set, 2002
19. Solingen R., Berghout E.: The Goal/Question/Metric Method; McGraw Hill Verlag, Berkeley, 1999

Modellierung von Software-Security mit aktivitätenbasierten Qualitätsmodellen

Stefan Wagner, Shareeful Islam

Fakultät für Informatik
Technische Universität München
{wagnerst,islam}@in.tum.de

Zusammenfassung *Security* oder *Informationssicherheit* stellt für Software immer noch eine große Herausforderung dar. Trotz breiter Anstrengungen Software sicher zu machen, ist die Zahl der berichteten Schwachstellen unvermindert hoch. Um dem entgegenzuwirken ist es wichtig, Security-Anforderungen klar zu formulieren und den Entwicklern und der Qualitätssicherung detaillierte Richtlinien an die Hand zu geben. Dazu wird die Modellierung von Software-Security mit Hilfe von aktivitätenbasierten Qualitätsmodellen vorgestellt.

1 Einleitung

Trotz der vielfältigen und kostspieligen Anstrengungen, die Sicherheit von Software sicherzustellen, ist bei den öffentlich-gemachten Sicherheitslücken kein signifikanter Rückgang zu verzeichnen [1]. Darüberhinaus sind aber nicht nur solch einzelne Lücken, sondern auch das Zusammenspiel aller Security-Mechanismen in einem Software-System entscheidend, was die Komplexität der Sicherheitsanalyse weiter erschwert. Software-Security ist daher immer noch eine große Herausforderung in heutigen Softwaresystemen.

Qualitätsmodelle beschreiben was mit *Qualität* bedeutet und verfeinern dieses Konzept in einer strukturierten Art und Weise. In der Terminologie von [2] sind also Qualitätsdefinitionsmodelle gemeint. In der Praxis wird dies oft auf Metriken wie *Zahl der gefundenen Fehler* oder sehr abstrakte Beschreibungen wie in der ISO 9126 [3] reduziert. Grundsätzlich werden Qualitätsmodelle mindestens auf zwei Arten in einem Softwareprojekt eingesetzt: (1) als Basis zur Definition von Qualitätsanforderungen und (2) zur Zuordnung von qualitätssichernden Maßnahmen und Messungen zu den Qualitätsanforderungen. Ersteres wird in der Regel durch die Beschränkung üblicher Qualitätsattribute (Zuverlässigkeit, Wartbarkeit, ...) eines Qualitätsmodells erreicht. In der Praxis findet man beispielsweise verkürzte Formulierungen wie "Das System soll einfach wartbar sein." Die zweite Verwendung von Qualitätsmodellen wird oft nicht explizit durchgeführt, sondern Metriken, wie die Zahl der durch Inspektion und Test gefundenen Fehler, werden direkt verwendet. Der Grund ist die hochkomplexe Zuordnung von Metriken zu abstrakten Qualitätsattributen. Es ist also prinzipiell wünschenwert, wenn Qualitätsmodelle hier mehr Struktur und Detail liefern, so dass sie eng in den Entwicklungsprozess integriert werden können.

Problemstellung Security sollte, wie auch andere Qualitätsattribute, früh im Entwicklungsprozess berücksichtigt werden. Jedoch fehlen auch im Bereich der Sicherheit immer noch integrierte und konkrete Qualitätsmodelle, die sowohl die präzise Spezifikation von Sicherheitsanforderungen, als auch deren direkte Umsetzung und Überprüfung im System unterstützen.

Beitrag Das erprobte Vorgehen, Qualität mit Hilfe von aktivitätenbasierten Qualitätsmodellen zu beschreiben, wird auf Security übertragen. Dabei werden bekannte Quellen benutzt und integriert. Ziel ist die Integration von Security mit Hilfe der Modellierung von System, System-Elementen, Umgebung und Prozess und deren Einfluss auf Aktivitäten, insbesondere Angriffe. Dadurch wird die Sicherstellung von Security mit in den allgemeinen Qualitätsmanagement-Prozess eingebunden.

2 Aktivitätenbasierte Qualitätsmodelle

Zur Erreichung dieser Strukturierung und des Detailgrads wurde die Verwendung von aktivitätenbasierten Qualitätsmodellen vorgeschlagen [4]. Die Idee ist, abstrakte "-ilities" für die Definition von Qualität zu vermeiden und stattdessen Qualität in detaillierte Fakten herunter zu brechen und deren Einfluss auf die Aktivitäten, die auf und mit dem System durchgeführt werden, zu beschreiben. Für Wartbarkeit wird dies beispielsweise in [4] gezeigt. Das Modell enthält Informationen über die Charakteristika eines Softwaresystems und anderer interessanter Umgebungseinflüsse und deren Einfluss auf Wartungsaktivitäten, wie beispielsweise Code lesen, Modifizieren oder *Testen*. Ein konkretes Beispiel dafür sind redundante Methoden im Quelltext, also so genannte Klone. Sie haben verschiedene Einflüsse, besonders wichtig ist aber der negative Einfluss auf Modifikationen des Quelltextes, da Änderungen an Klonen an verschiedenen Stellen im Text durchgeführt werden müssen. Das bedeutet, dass falls eine Systementität Methode das Attribut REDUNDANZ besitzt wird dies einen negativen Einfluss auf die Aktivität Modifizieren (also eine Änderung der Methode) haben.

Das Modell enthält nicht nur diese Einflüsse der Fakten auf Aktivitäten sondern auch die Zusammenhänge untereinander. Sowohl die Fakten als auch die Aktivitäten sind als Hierarchien abgelegt. Die oberste Aktivität Aktivität besitzt Unteraktivitäten wie Benutzen, Warten oder Administrieren (siehe auch Abb. 1). In praxistauglichen Modellen werden diese dann weiter verfeinert. Beispielsweise kann die Wartung die Unteraktivitäten Code lesen und Modifizieren haben.

Die Fakten setzen sich zusammen aus *Entitäten* und *Attributen*, also Charakteristika einer Entität. Dies erlaubt die Entitäten einfach in einer Hierarchie zu organisieren. Die oberste Ebene ist hier die Situation des Softwareentwicklungsprojekts. Es enthält beispielsweise das System, seine Umgebung und die Entwicklungsorganisation. Wiederum müssen die Entitäten weiter verfeinert werden. Beispielsweise besteht das System aus statischen und dynamischen Aspekten. All Entitäten werden durch Attribute beschrieben. Ein Beispiel-Fakt ist die STRUKTURIERTHEIT des Systems: Ist das System in einer sinnvollen und definierten Art und Weise strukturiert? Diese Fakten sind entweder automatisch oder manuell überprüfbar. Soweit möglich werden auch entsprechende Metriken angegeben.

Beide Hierarchien, der Faktenbaum und der Aktivitätenbaum, können zusammen mit den Einflüssen der Fakten auf die Aktivitäten als Matrix wie in Abb. 1 dargestellt werden. Die Einflüsse sind dabei als Einträge in die Matrix dargestellt, wobei ein "+" ein positiver und ein "-" ein negativer Einfluss ist.

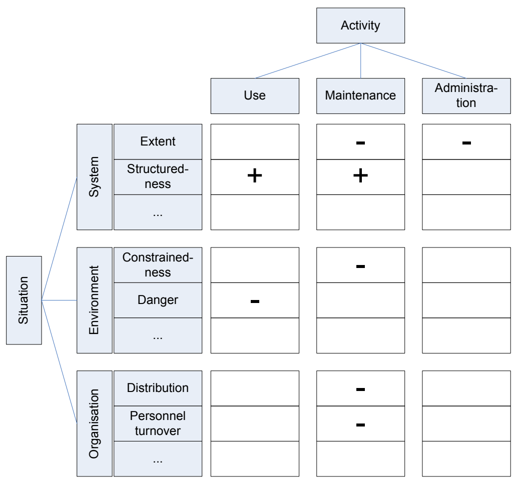


Abbildung 1. Abstrakte Darstellung eines aktivitätenbasierten Qualitätsmodells als Matrix

Es existiert eine abstrakte, textuelle Notation, die die entsprechenden Entitäten, ihre Attribute und deren Einfluss auf Aktivitäten kompakt darstellt. Beispielsweise ist das folgende Tupel ein Eintrag im Qualitätsmodell über konsistente Bezeichner: $[\mathsf{Bezeichner} \mid \mathsf{KONSISTENZ}] \xrightarrow{+} [\mathsf{Modifizieren}]$ Dies bedeutet, dass konsistente Bezeichner einen positiven Einfluss auf Modifikationen am System haben. Im Modell selbst werden darüberhinaus aber noch weitere Informationen, wie ausführliche Beschreibungen und Quellen, dokumentiert.

Durch die Verwendung aktivitätenbasierte Qualitätsmodelle können früh konkrete und überprüfbare Anforderungen aufgestellt [5] und später auch konkrete Qualitätssicherungsmaßnahmen abgeleitet werden. Beispielsweise können Checklisten für Reviews automatisch generiert werden [4]. Weitere Informationen über diese Art von Modellen und den bisherigen Erfahrungen in der Praxis sind in [4, 6–8] zu finden.

3 Das Security-Modell

Das wichtigste neue Konzept für die Modellierung von Security ist Angriffe als Aktivitäten zu sehen und in den Aktivitätenbaum zu integrieren. Diese Aktivitäten müssen also durch Fakten negativ beeinflusst werden, um ein hochqualitatives System zu erreichen.

3.1 Der Aktivitätenbaum

Zuvorderst muss zwischen erwarteten und unerwarteten Angriffen unterschieden werden. Eine Hauptschwierigkeit bei Security ist nämlich, dass es unmöglich ist, alle zukünftigen Angriffe, dem ein System ausgesetzt sein wird, zu kennen, da täglich neue Angriffe entwickelt werden. Deshalb ist es wichtig zwei Strategien zu verfolgen: (1) das System gegen erwartete Angriffe absichern und (2) das System prinzipiell zu härten, also unempfindlicher gegenüber Angriffen zu machen. Für die Klassifizierung der Angriffe können verschiedene Quellen verwendet werden. Wir bauen hauptsächlich auf die *Common Attack Pattern Enumeration and Classification* (CAPEC) [9], die vom U.S. Department of Homeland Security vorangetrieben wird. In CAPEC werden existierende Angriffsmuster gesammelt und klassifiziert. Diese Angriffe werden in einer Hierarchie angeordnet, die direkt im Aktivitätenbaum wiederverwendet werden kann (Abb. 2).

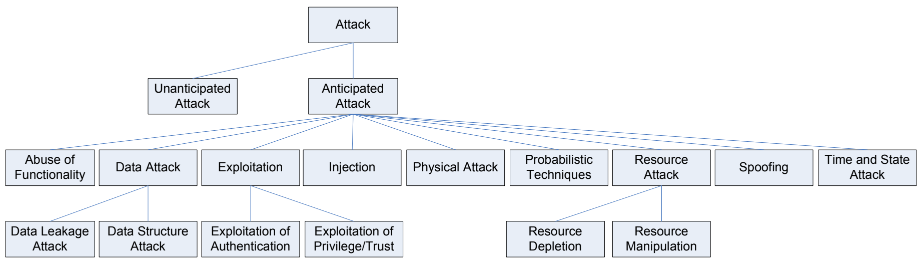


Abbildung 2. Die obersten Ebenen des Angriff-Teilbaums des Aktivitätenbaums

Die erwarteten Angriffsarten (*anticipated attacks*) werden in eine Reihe von Mustern und Untermustern aufgeteilt. Die Einteilung folgt dabei im wesentlichen der CAPEC und ist dabei nicht völlig überscheidungsfrei. Dies ist durch die Komplexität der Angriffsarten nicht möglich, aber auch nicht notwendig. Entscheidend ist, dass bestimmte Angriffsmuster schnell gefunden werden können. Teilweise wird dabei unterschieden, *was* angegriffen wird (*Abuse of Functionality*, *Data Attack*, *Physical Attack*, *Resource Attack*), teilweise auch *wie* der Angriff durchgeführt wird (*Exploitation*, *Injection*, *Probabilistic Technique*, *Spoofing*, *Time and State Attack*).

3.2 Der Faktenbaum

Die Erstellung des Faktenbaums ist demgegenüber wesentlich komplizierter. Es muss das vorhandene Wissen über Charakteristiken eines Systems, der Umgebung und Organisation enthalten, die diese Angriffe beeinflussen. Dazu verwenden wir wiederum eine Reihe von Quellen, unter anderem die ISO/IEC 27001 [10] oder die Sun Secure Coding Guidelines for the Java Programming Language [11]. Zwei Hauptquellen stellen dabei bestimmte Teile der *Common Criteria* (CC) [12] und die *Common Weakness Enumeration* (CWE) [13] dar. Die Common Criteria beschreiben Anforderungen, was ein System leisten soll, damit es sicher ist. Dabei liegt der Fokus auf Funktionalitäten. Die CWE beleuchtet dies stärker von der anderen Seite und beschreibt wiederkehrende Schwachstellen, die von Angriffen ausgenutzt wurden. Zusammengenommen liefern diese beiden Quellen ein starkes Fundament für das Security-Modell.

Wir können hier nicht alle Details des Modells darstellen, geben aber einige Beispiele, wie Wissen aus den Quellen in das Modell überführt wurde. Dafür verwenden wir einen Unterbaum des Faktenbaums (Abb. 3).

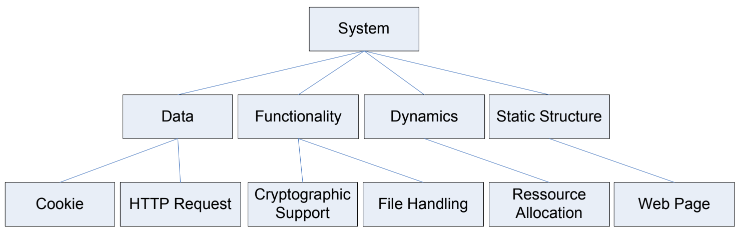


Abbildung 3. Beispielhafte Einträge des Faktenbaums

Viele der Einträge im Qualitätsmodell, die ihren Ursprung in den CC haben, wurden als Teil der Entität Functionality modelliert, da sie überwiegend Verhaltensaspekte beschreiben, die für Security wichtig sind, beschreiben. Ein Beispiel ist die kryptographische Unterstützung des Systems als Teil der Funktionalität. Nach den CC kann sie in Cryptographic Key Management und Cryptographic Operation unterteilt werden. Ersteres enthält wiederum Cryptographic Key Generation, zu dem in den CC definiert wird, dass sie in Übereinstimmung mit einem spezifizierten Algorithmus und spezifizierten Schlüsselgrößen sein soll. Im Modell wird dies durch die Verwendung des Attributes APPROPRIATENESS für Cryptographic Key Generation ausgedrückt. Die textuelle Beschreibung dieses Fakts ist dann: “The system generates cryptographic keys in accordance with a specified cryptographic key generation algorithm and specified cryptographic key sizes that meet a specified list of standards.” Die CC enthalten leider keine Beschreibungen von Einflüssen, die sie noch nützlicher machen würden, da die Motivation für die angegebenen Anforderungen gleich mitgeliefert würden. Aus diesem Grund ergänzen wir diese Information aus anderen Quellen. In diesem Fall enthält CAPEC die Beschreibung des Angriffs *Kryptoanalyse* und die Vermeidungsstrategie bewährte kryptographische Algorithmen mit empfohlenen Schlüsselgrößen zu verwenden. Also sieht der Einfluss wie folgt aus: [Cryptographic Key Generation | APPROPRIATENESS] $\xrightarrow{-}$ [Cryptanalysis]

Die CWE enthält oft Charakteristiken eines Systems und speziell von Quelltext, die vermieden werden sollten. Dies kann so im Modell verwendet werden.

Einige beschriebene Schwachstellen in CWE beziehen sich nicht auf bestimmte Attacken, werden aber als Indikatoren für potenzielle Sicherheitslöcher angegeben. Diese werden als Fakten modelliert, die einen Einfluss auf unerwartete Angriffe haben. Ein Beispiel dafür ist toter Code, beispielsweise ungenutzte Variablen. Im Modell ist dies wie folgt enthalten: [Variable | SUPERFLUOUSNESS] $\xrightarrow{+}$ [Unanticipated Attack].

Schließlich muss noch angemerkt werden, dass im Modell nicht nur die Möglichkeit besteht Einflüsse auf Angriffe zu modellieren, sondern auch andere Aktiviäten, wie die Benutzung des Systems können beeinflusst werden. Abhängig von der Definition von Security kann man damit beispielsweise auch die zufällige Veröffentlichung sensibler Daten darstellen.

3.3 Beispiel

Wir bearbeiten gerade eine größere Fallstudie zur Anwendung des Security-Modells auf den Servlet-Container Tomcat[1]. Ingesamt liegt der Fokus dort auf der Bestimmung der Security-Anforderungen. Wir beschränken uns hier aber nur auf den Vergleich der veröffentlichten Security-Lücken von Tomcat und korrespondierender Einträge im Qualitätsmodell. Dies erlaubt noch keine quantitative Aussage über den Wert des Einsatzes des Qualitätsmodells, aber es zeigt das Potential des Ansatzes auf.

Insgesamt wurden für die Tomcat-Version 6.0 vom Dezember 2006 bis Juli 2008 19 Schwachstellen veröffentlicht. Die Tomcat-Entwickler klassifizieren diese Schwachstellen in die Typen *Cross-Site Scripting*, *Session Hi-Jacking*, *Directory Traversal*, *Information Disclosure*, *Data Integrity* und *Elevated Privileges*. Diese Typen können also zum Teil bereits auf unseren Aktivitätenbaum zurückgeführt werden. Weiterhin wurde die Wichtigkeit der Schwachstellen bewertet. Es wurden nun alle diese Schwachstellen mit dem Security-Modell verglichen, um zu analysieren, ob entsprechende Qualitätsregeln enthalten sind. Tabelle 1 zeigt die Zahl der Schwachstellen im Verhältnis zur Zahl der entsprechend gefundenen Qualitätsregeln im Modell auf. Es zeigt sich, dass nur für drei von 19 Schwachstellen keine Regel gefunden werden konnte. Es hätte also die Chance bestanden durch Verwendung des Modells diese Schwachstellen zu vermeiden. Ob dies wirklich gelingen kann, wird derzeit in der Fallstudie untersucht.

4 Verwandte Arbeiten

Allgemeine Qualitätsmodelle, die einen Dekompositionsmechanismus verwenden wurden bereits von Boehm [14] und McCall [15] vorgeschlagen. Trotz der vielfältig dokumentierten Schwächen [16] folgen aktuelle ISO-Standards immer noch dieser Tradition [3]. Im Gebiet der Security, können die oben erwähnten Richtlinien wie die *Secure Coding Guidelines for the Java Programming Language* oder die *CERT C Secure Coding Standard* verwendet werden. Solche Richt-

[1] `http://tomcat.apache.org/`

Tabelle 1. Zusammenfassung des Vermeidungspotentials

Typ	**Anteil**	**Wichtigkeit**	**Anteil**
cross-site scripting	7/7	low	10/10
session hi-jacking	4/4	moderate	2/2
directory traversal	1/2	important	4/7
information disclosure	3/4	total	16/19
data integrity	0/1		
elevated privileges	1/1		

linien sind wichtig und wertvoll, da sie spezifische und konkrete, oft auch überprüfbare Regeln angeben. Jedoch enthalten sie meist nicht den Einfluss, den die Ein- bzw. Nichteinhaltung der Regeln hat. Darüberhinaus liefern Security-Normen, wie die ISO/IEC 27001 [3] und die *Common Criteria* [12] einen breiteren Blick auf Security. Es werden nicht nur fr Quelltext Richtlinien angegeben, sondern auch Aspekte der Funktionalität oder der Organisation berücksichtigt. Dies führt aber auch dazu, dass diese Normen wesentlich generischer und damit schwieriger zu überprüfen sind. Auch hier werden Auswirkungen und Einflüsse kaum berücksichtigt.

Chung und Nixon [17] haben eine systematische Methode zum Umgang mit Qualitätscharakteristika, das NFR-Rahmenwerk. Dieser Ansatz betrachtet Korrelationsregeln, um existierende Ziele mit neuen Zielen zu verbinden. Eine Einbindung in die Qualitätssicherung wird aber nicht explizit angegeben. Zu einem gewissen Grad, stehen auch Anforderungsansätze für Security-Anforderungen im Bezug zu dieser Arbeit. Bekannte Vertreter sind hier SQUARE [18] und darauf aufbauend SREP [19]. Beide konzentrieren sich stärker auf den Prozess, beziehen sich aber sonst auf klassische Qualitätsmodelle.

5 Zusammenfassung

Die hier beschriebene Modellierung von Security mit Hilfe von aktivitätenbasierten Qualitätsmodellen stellt ein strukturiertes Vorgehen zur Erfassung von sicherheitsrelevantem Wissen zur Verfügung. Darüberhinaus kann so auch die Sicherstellung von Sicherheit in den normalen Qualitätsmanagement-Prozess eingebunden und sogar Überlappungen mit anderen Qualitätsattributen identifiziert werden. Wir arbeiten derzeit an einem umfassenden *Security Requirements Engineering*-Ansatz auf Basis des beschriebenen aktivitätenbasierten Qualitätsmodells.

Danksagung

Diese Arbeit entstand teilweise mit Förderung des Bundesministeriums für Bildung und Forschung (BMBF) im Projekt *QuaMoCo* (Förderkennzeichen: 01 IS 08023B).

Literatur

1. CERT: Vulnerability remediation statistics. Online verfügbar unter `http://www.cert.org/stats/vulnerability_remediation.html`
2. Deissenboeck, F., Juergens, E., Lochmann, K., Wagner, S.: Software quality models: Purposes, usage scenarios and requirements. In: Proc. 7th International Workshop on Software Quality (7-WoSQ), IEEE Computer Society (2009)
3. ISO: ISO 9126: Product Quality – Part 1: Quality Model (2003)
4. Deissenboeck, F., Wagner, S., Pizka, M., Teuchert, S., Girard, J.F.: An activity-based quality model for maintainability. In: Proc. 23rd International Conference on Software Maintenance (ICSM '07), IEEE Computer Society Press (2007)
5. Wagner, S., Deissenboeck, F., Winter, S.: Managing quality requirements using activity-based quality models. In: Proc. 6th International Workshop on Software Quality (WoSQ '08), ACM Press (2008) 29–34
6. Winter, S., Wagner, S., Deissenboeck, F.: A comprehensive model of usability. In: Proc. Engineering Interactive Systems 2007 (EIS '07). Volume 4940 of LNCS., Springer (2008)
7. Wagner, S., Deissenboeck, F., Feilkas, M., Juergens, E.: Software-Qualitätsmodelle in der Praxis: Erfahrungen mit aktivitätenbasierten Modellen. In: Workshop-Band SQMB 2008, TU München (2008)
8. Mas y Parareda, B., Streit, J.: Software quality put into practice. In: Workshop-Band SQMB 2008, TU München (2008)
9. Homeland Security: Common attack pattern enumeration and classification (CAPEC). Available Online at http://capec.mitre.org/. Accessed in October 2008
10. ISO: ISO/IEC 27001: Information technology – Security techniques – Information security management systems – Requirements (2005)
11. Microsystems, S.: Secure coding guidelines for the java programming language, version 2.0. Available Online at `http://java.sun.com/security/seccodeguide.html`
12. CCRA: Common criteria for information technology security evaluation, version 3.1. Available Online at `http://www.commoncriteriaportal.org/`
13. Homeland Security: Common weakness enumeration (CWE). Available Online at http://cwe.mitre.org/. Accessed in October 2008
14. Boehm, B.W., Brown, J.R., Kaspar, H., Lipow, M., Macleod, G.J., Merrit, M.J.: Characteristics of Software Quality. North-Holland (1978)
15. McCall, J.A., Richards, P.K., Walters, G.F.: Factors in software quality. Reports NTIS AD/A-049 014, 015, 055, US Rome Air Development Center (1977)
16. Kitchenham, B., Pfleeger, S.L.: Software quality: The elusive target. IEEE Software **13**(1) (1996) 12–21
17. Chung, L., Nixon, B.A.: Dealing with non-functional requirements: Three experimental studies of a process-oriented approach. In: Proc. 17th ICSE. (1995) 25–37
18. Mead, N., Steheny, T.: Security quality requirement engineering methodology. In: Proc. Workshop on Software Engineering for Secure Systems (SESS '05). (2005)
19. Mellado, D., Medina, E., Piattini, M.: Acommon criteria based security requirements engineering process for the development of secure information system. Computer standards & interfaces **29** (June 2007) 244–253

Towards an applicable software quality model for individual software projects

Markus Großmann

Capgemini sd&m AG, Löffelstr. 46, 70597 Stuttgart, Germany
markus.grossmann@capgemini-sdm.com

Abstract. Capgemini sd&m develops quality models and tools for controlling the software product quality and has been applying them in their individual software projects since a couple of years. The first version of the quality model was an attempt to implement ISO 9126-1 and the GQM method. It appeared that the desired overall software quality characteristics (e.g., maintainability) couldn't be measured sufficiently. "Magic" quality index values were distrusted. Projects had difficulties at applying GQM. It showed that there was no need for a flexible, sophisticated quality metamodel. This resulted in a change of the quality model design: the current version of the quality model took up the idea of a "software blood count", i.e., a collection of software product indicators relevant to quality assessment. Quality is not automatically computed but instead rated by an experienced person that knows the project context and uses the blood count as additional information. The software blood count contains only few, in practice relevant metrics, together with heuristics and interpretation patterns/antipatterns for analyzing quality risks. First practical experiences with individual software projects showed that such a lightweight quality model was better accepted and could be easier applied.

1 Introduction

Recent evolutions in the area of commercial software development put software quality under pressure. There is a high time and cost pressure together with shorter release cycles and stiff competition. The systems to be developed become larger and more complex. Further challenges come with offshore development, SOA and the integration of individual and standard software. This all demands an increased effort and precise controlling to ensure a high quality of the produced software.

However, in comprehensive projects the assessment and controlling of the software quality is particularly difficult. Quality requirements are often not precisely defined or not explicitly requested. Means for assuring the "inner" quality characteristics of the software (e.g., maintainability) are often not in place.

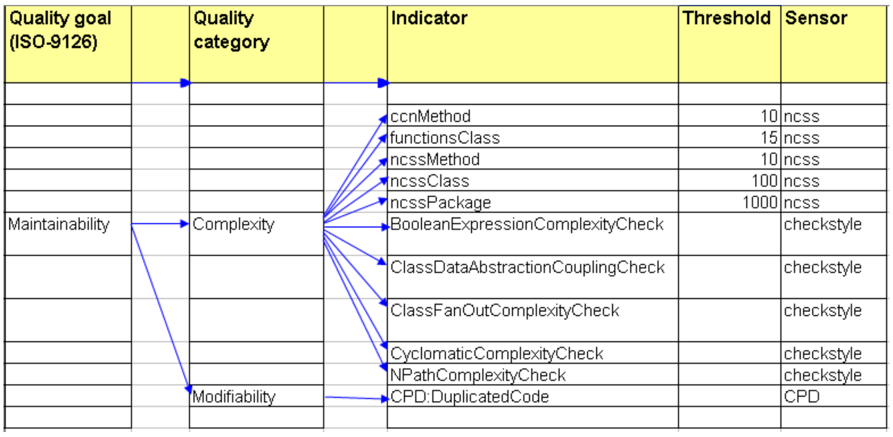
A high quality of software development projects and of the built products is an important business objective of Capgemini sd&m. Therefore Capgemini sd&m developed several constructive and analytical quality assurance techniques in the past and used them in their software projects. Among them are analytical tools for the assessment and visualization of product quality, centralized within a tool called “software cockpit” and based on a quality model which was developed together with a research partner [1].

Quality models have become a well-accepted means to describe and manage software quality. However, there are still problems with them so that they haven’t become widely adopted in practice. In the following the problems and experiences are summarized which we faced at Capgemini sd&m when working with quality models in individual software projects. We also depict a possible solution for an in practice applicable quality model.

2 Experiences with the ISO 9126 quality model and GQM

The first version of the Capgemini sd&m quality model [1] was very much inspired by the quality model of ISO 9126 [2] and the Goal-Question-Metric method [3]. The motivation was to have a standardized quality model that measures overall quality characteristics of the developed software (e.g., maintainability). Furthermore it should be possible that projects create their own quality model for their specific quality goals. Therefore the software cockpit provided a flexible quality metamodel that could be instantiated by projects and it provided indicators that were measured automatically by (only) static code analysis.

Additional input was a standard quality model, an instance of this quality metamodel, that contained on the top-level the goals from ISO like maintainability, efficiency and reliability. Projects should take the standard quality model, extend it with their own quality goals and adjust the indicators where necessary. Fig. 1 shows a fragment of the standard quality model. It also outlines the elements of the quality metamodel: quality goals further refined by quality categories. Each quality category could be assigned one or more quality indicators. With the GQM approach this kind of structuring suggests itself and will not be further explained here. Details about the quality metamodel can be found in [1].



Quality goal (ISO-9126)		Quality category		Indicator	Threshold	Sensor
				ccnMethod	10	ncss
				functionsClass	15	ncss
				ncssMethod	10	ncss
				ncssClass	100	ncss
				ncssPackage	1000	ncss
Maintainability		Complexity		BooleanExpressionComplexityCheck		checkstyle
				ClassDataAbstractionCouplingCheck		checkstyle
				ClassFanOutComplexityCheck		checkstyle
				CyclomaticComplexityCheck		checkstyle
				NPathComplexityCheck		checkstyle
		Modifiability		CPD:DuplicatedCode		CPD

Fig. 1. Example for the breakdown of a quality goal to quality indicators

The set of indicators for the first version of the standard quality model came from a handful of experienced software architects at Capgemini sd&m and from research done during the development of a method for controlling software product maturity [4]. This set of indicators was only a first serve. We expected a learning phase in which we take up feedback from the projects for further improvement of the standard quality model.

2.2 Problems and experiences

The first version of the software cockpit implemented the described quality model and provided a first version of the standard quality model. We tested it in six individual software projects which were in the domains of automotive, telecommunication and public sector. Both large and medium size projects were represented.

2.2.1 Problems with ISO 9126 and quality indexes

It appeared very soon that the quality characteristics of the ISO 9126 could not be measured sufficiently, for example the measurement of maintainability. Only indicators for maintainability could be measured, not maintainability itself. Both seemed to be correlated, however indicators could be manipulated by writing code in a way that it gives good indicator values and correlation vanished. Furthermore there were characteristics of source code that were highly relevant for maintainability but couldn’t be measured automatically (e.g., the understandability of a source code comment). So this method was scarcely suitable for obtaining a more precise control on maintainability. People distrusted “magic” result values like a maintainability index that was calculated from quality indicators. We experienced that such index values were also too fragile to be comparable between projects. E.g., some projects didn’t apply all quality indicators. In some projects misleading values were contributed to the index (false positives). From a practioners viewpoint there was no problem, but the quality index was impaired. It was also hard to determine whether the index value represented a good or a bad value regarding the expected maturity of the product. E.g., low values for maintainability needn’t be an indicator for bad quality when building prototypes. The project management was misled to use the index as a general measure and not as an indicator for quality. They wanted the index values to be as reliable as the other metrics they were dealing with (e.g., time and budget). “Use them but do not trust them” [5] was from a management point of view unacceptable.

Conclusion 1: An applicable quality model shouldn’t contain any quality indexes. It shouldn’t try to quantify software quality characteristics that can only be vaguely measured. The risk is too high that quality indexes get broken and “guess” wrong, are used as a simplistic answer for quality controlling or used to whitewash a project. It is far the worst when such quality indexes are used in form of traffic lights at project reporting. Sure, the demand for such an index is obvious, regarding the enormous influence of maintainability for the long term ownership costs of software. Though current quality models and the GQM approach appear to be no appropriate way to determine it in practice. They draw a picture but are not able to tell the full truth. It is like testing the intelligence of a person with questions that are already known in advance or if you want to determine the stability of a building solely by analyzing a picture of it.

2.2.2 Problems with GQM

We experienced that projects had difficulties at including their non-functional requirements into the quality model. Most of the requirements were formulated in a way that it was not obvious how to measure them (e.g., "the application should be scalable"). Hence people didn't know which of the quality indicators or metrics they should select. And when they agreed to a solution they were still not sure how good the quality of their quality model was. It's like a dog chasing its tail. This problem was not caused by a lack of experience with metrics but was a general problem: the freedom and flexibility of a generic quality metamodel made the configuration too complex. People in the projects didn't had the time to get acquainted with a complex quality model hence they didn't use it.

Conclusion 2: An applicable quality model should be as simple as possible. It should be less sophisticated and prohibit improper configuration. The challenge is to find a quality model that hits an optimum between a high benefit and a low effort. However, we also see the necessity for research at how to formulate non-functional requirements in a way that they can be measured in practice. As far as individual software projects today are too different here – some are more or less formal than others – GQM seems to be a not universally applicable approach.

2.2.3 Quality rating by using heuristics

We experienced that projects directly looked at the metric values – in particular at time trends. They used own heuristics to find potential quality problems (e.g., "Where should I refactor classes with high cyclomatic complexity?", "How is the trend of the number of code duplicates developing?"). Here it showed that experienced people knew more heuristics, had more ideas and better interpretation explanations than inexperienced people. Experienced people were able to bring in their minds together the quality indicator values, the project context and their knowledge about the project-specific correlations. Most of the architects and projects leaders in our observed projects had this experience. With experience we mean broad software engineering know-how together with project experience.

We also saw that the overview got lost if metric values were collected on a fine grained class and method level. Most people regarded it as sufficient to collect metrics on a coarse grained component level. The first version of the software cockpit included only tools for static code analysis. People encouraged us to include more information sources related to quality assurance means (e.g., defects and test results).

Conclusion 3: An applicable quality model should consider heuristics and interpretation patterns to be a central topic. Higher aggregations i.e. overall quality ratings should only be done by experienced people. However, the quality assessment should be supported by a couple of known patterns and antipatterns for quality indicators.

Conclusion 4: An applicable quality model should provide metric values on a component and subsystem level and should not focus only on source code metrics. It should try to integrate as many as possible different information sources that give answers to quality related questions.

2.2.4 Only collecting metrics is not enough

We experienced that it was not enough to collect the metrics and leave it to the project to think about consequences or use cases where they could use the metrics. The use cases of the software cockpit where not obvious, which made it difficult to give projects clues about application areas of the quality model. Projects interpreted the quality indicators but had difficulties at guessing appropriate consequences. Sure, they often had the right ideas about how to react on a specific quality risk. But the process was little structured and not very straightforward.

Conclusion 5: An applicable quality model should embed the quality indicators in a workflow that starts with a use case and ends with clues about possible consequences.

3 The Capgemini sd&m software blood count

The conclusions and experience with the first version of the quality model led us to a new design of the quality model and a new philosophy how to use it. The idea: a "software blood count" shall support the rating of the software quality. The analogy comes from medicine: when your family doctor wants to check your health he doesn't pull out his pocket calculator and computes your health based on your size or your weight. He will rather ask you some questions, auscultate your lungs and your heartbeat and look at your blood count. The blood count may contain some risk factors which your doctor brings in his mind together with the other information before he lets you know his appraisal of your health.

The software blood count is a collection of few software product statistics which are considered to be in practice relevant for the assessment of quality. Embedded in an analytical workflow and together with interpretation patterns for these metrics it forms a quality model where the quality is rated by an experienced person and not calculated by a computer.

3.1 The role of quality indicators

The general design of the new quality model is shown in Fig. 2.

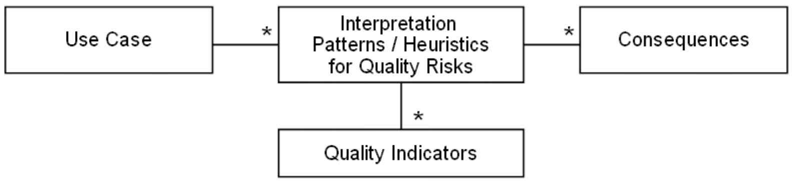


Fig. 2. Structure of the new quality model

At first sight there seem to be some similarities[1] between the first version of the quality model and the new version. However, the role of quality indicators has changed. Watching quality indicators helps projects to recognize early whether they are leaving the path of good quality. So the quality models aim is to reveal quality risks. Our experience is that is easier to measure "bad smells" of software or a project than to attest a high quality level. The benefit is anyhow considerable: projects can adjust the development process before quality deficiencies start to pollute it. The quality model indicates whether things are going well, or more analysis should be performed, or whether the process needs to be adjusted because it is on the wrong way. Today, such decisions often are either taken by gut feeling, or when trouble has become obvious and projects are already in a state that they would have wanted to avoid.

3.2 Examples for the implementation of the quality model

The software cockpit is the tool that automatically collects and displays the quality indicators. This is essential, because otherwise collecting data is a too cumbersome task to be performed regularly. It tells which indicators to watch and how to interpret them. Currently the cockpit structures the interpretation patterns by the following use cases, which appear directly in the GUI:

- Check the current build of the software
- Define and check quality guidelines
- Observe trends which may lead to quality problems
- Find quality assurance blind spots
- Find outliers and spots with potential quality problems

Each use case is associated with some patterns that support the interpretation. They contain descriptions of quality risks and how they usually become manifest in the quality indicator values. These patterns aim at revealing quality risks that are associated with significant, avoidable costs. So, e.g., empty exception handlers are normally no quality risk. The following table shows some examples for quality heuristics that are currently implemented in the software cockpit. At present there are about 30 patterns and heuristics available in the software cockpit. Users are free in their choice to directly analyze and explore quality indicators or to go through the patterns.

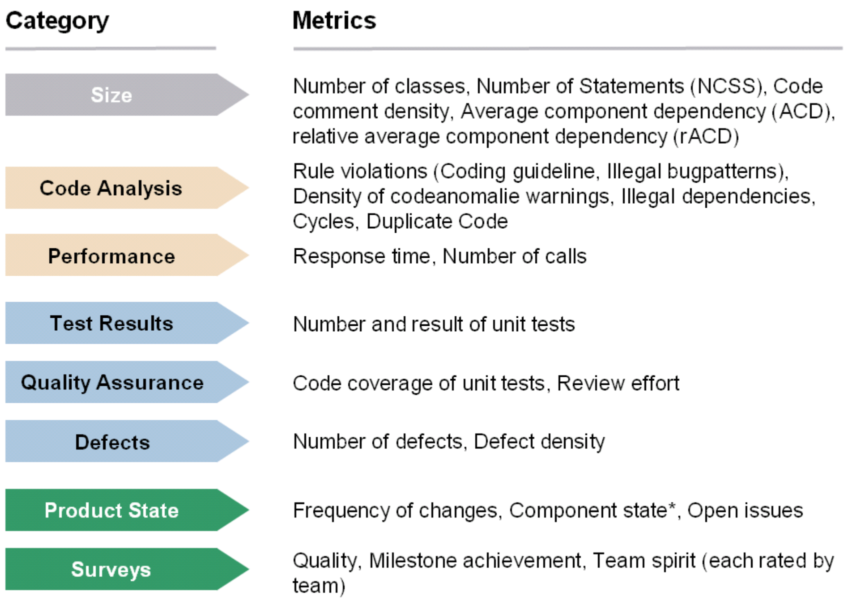
[1] E.g., you could replace "Use Case" with "Goal" and "Categories" with "Heuristics"

Use Case	Quality heuristics / patterns	Metrics	Consequences
Observe risky trends	The coupling grows and grows	Average component dependency	Analyze code dependencies, plan refactoring for removing unwanted dependencies
Observe risky trends	Amount of tests is decreasing	Number of unit tests, Code coverage of unit tests	Analyze reason of the decrease
Observe risky trends	Team in "Hacking mode": the code anomaly density grows stronger than the size of code	Code anomaly density, NCSS	Check whether the team doesn't care about warnings and tries to produce code as fast as possible. Team may need more time for coding.
Check the current build	Code contains architecture violations	Illegal dependencies	Remove architecture violations as soon as possible
Check the current build	Test suite is broken	Number of failed tests	Analyze failed tests and repair
Check quality guidelines	Coding standard overboard	Number of violated coding style rules	Reformat code according to your coding guideline, check tool support to assert code style

Table 1. Examples for quality heuristics

3.2 Metrics of the software blood count

The software blood count contains source code attributes, defect statistics, performance numbers as well as review and testing results. We also included some soft factors, e.g., survey results. The software cockpit provides more than 20 metrics, with a time trend and a component comparison for each.



Category	Metrics
Size	Number of classes, Number of Statements (NCSS), Code comment density, Average component dependency (ACD), relative average component dependency (rACD)
Code Analysis	Rule violations (Coding guideline, Illegal bugpatterns), Density of codeanomalie warnings, Illegal dependencies, Cycles, Duplicate Code
Performance	Response time, Number of calls
Test Results	Number and result of unit tests
Quality Assurance	Code coverage of unit tests, Review effort
Defects	Number of defects, Defect density
Product State	Frequency of changes, Component state*, Open issues
Surveys	Quality, Milestone achievement, Team spirit (each rated by team)

Fig. 3. The Capgemini sd&m software blood count

3.4 Conclusion and outlook

Our first experiences at applying the new quality model in projects are promising. Coming back to our five conclusions from the first quality model we are confident to have addressed all relevant weaknesses and made a step forward towards an applicable quality model for individual software projects. The software blood count as a lightweight and easy to understand quality model was better accepted by projects. The setup time was minimal (few days), the learning effort low – people could work with the software cockpit in an intuitive way. Quality appraisal with the software cockpit is a matter of hours, not a matter of days. Up to now the flexibility of a configurable quality metamodel was not needed.

Quality indicators are not a measure of software quality. There is no simple measure for it. Our profound credo is that quality arises by the careful, creative work of a motivated team, not by tuning software metrics. The assessment of quality affects the human, not the software - though the human is supported by the software in the best possible manner. The software cockpit takes care of transparency over the maturity of software, serves as an early warning system for quality problems and helps – in accordance with the principle "knowing instead of believing" – to substantiate intuition with facts. It is not an autopilot but it permits systematic and continuous control of quality, as this was the case in the past.

Further work has to be done to collect and refine interpretation patterns and quality heuristics. This includes also a classification and a prioritization of quality risks. Moreover we want to do some empirical studies on how often the software blood count should be watched and examine which are the non-obvious quality risks that could be detected by the quality model.

References

[1] M. Bennicke, F. Steinbrückner, M. Radicke, J.-P. Richter: Das sd&m Software Cockpit: Architektur und Erfahrungen, in R. Koschke, O. Herzog, K.-H. Rödiger, M. Ronthaler (Hrsg.): INFORMATIK 2007, Beiträge der 37. Jahrestagung der Gesellschaft für Informatik e.V. (GI), Lecture Notes in Informatics (LNI), GI Proceedings 110, Band 2, pp. 254—260 (2007)

[2] ISO 9126-1 Software Engineering – Product quality – Part 1: Quality model (2003)

[3] V. Basili, G. Caldiera, H. D. Rombach: The Goal Question Metric Approach. In: Encyclopedia of Software Engineering, pp. 528—532, John Wiley & Sons (1994)

[4] H. Blau, S. Eicker, A. Hofmann, T. Spies: Reifegradüberwachung von Software. ICB Research Report No 20, ISSN 1860-2770 (2007)

[5] C. Lewerentz, H. Rust, F. Simon: Quality – Metrics – Numbers – Consequences, in R. Dumke, F. Lehner (Hrsg.): Software-Metriken, Entwicklungen, Werkzeuge und Anwendungsverfahren, Wiesbaden: Gabler (2000)

An Approach for a Method and a Tool Supporting the Evaluation of the Quality of Static Code Analysis Tools

Reinhold Plösch, Alois Mayr, Gustav Pomberger, Matthias Saft

(1) Johannes Kepler University Linz, Institute for Business Informatics – Software Engineering, Linz, Austria
reinhold.ploesch | alois.mayr1 | gustav.pomberger @jku.at
(2) Siemens AG, Corporate Technology – SE 1, Otto-Hahn-Ring 6,
81739 Munich, Germany
matthias.saft@siemens.com

Abstract. There is a lack of information concerning the quality of static code analysis tools. In order to overcome this we therefore developed a method and a tool supporting quality engineers to determine the quality of static code analysis tools. This paper shows how the method works and where the tool supports it. We already applied the combination of the method and its tool to two static code analysis tools in different versions. On this basis, we further illustrate some results of the usage of the method.

Keywords: static code analysis tool, PMD, CodePro AnalytiX, quality model, true positive, false positive, false positive rate.

1. Introduction and Overview

During software development, testing has high priority. One way to test software and evaluate its quality is to perform a dynamic test. The source code of the software product is executed in order to test the fulfillment of its requirements. On the other hand performing a static test does not need the execution of the source code, so static testing can be applied already in early development phases. Typically, these tools check the source code in order to indicate problem areas or possible flaws. There are a number of tools available (e.g. PMD [7], FindBugs [5], and CodePro AnalytiX [2] for the programming language Java) which possess different levels of quality regarding their results.

In this respect, quality of the results focuses on their preciseness. Of course, the quality of static code analyzers has more aspects than the preciseness of its results, e.g. the usability or performance. However, this paper uses the term quality in the meaning of the preciseness of the results of the analyzers as that is its scope. All other quality aspects remain ignored in this paper. Overall, the principle goal is to achieve statements about the quality of the results, in order to support the selection procedure of tools in practice. For this, we need to know the quality of the tools to allow proper usage.

There are only a few literatures, which covers the quality of static code analyzers. Some authors focus on the comparison of tools, e.g. [9] and [11], including the tool PMD. Therefore, they classify the output of the tools into various categories. In [11] the authors furthermore compare static code analysis with dynamic testing and informal reviews.

In order to be able to ascertain and subsequently to compare the quality of the results of static code analyzers or different versions of them, we need a method that supports the process of ascertaining their quality in a systematic way. Beside this method, we need a tool that automates as much tasks as possible to reduce manual work.

The goal of this paper is to present a method and a tool helping users to evaluate the quality of static code analyzers. Furthermore, it shows the application of the method in practice. Moreover, comparing the results, so-called findings, of some static code analyzers is possible. Furthermore, this paper describes the functionality of a tool helping to automatically rate findings obtained by the application of a static code analysis tool.

Both, the method and the tool enable determining the quality of several versions of static code analyzers easily. On this basis, you can identify the development of the quality of tools like their improvements or degradations over time (versions). Additionally, we want to find out about differences in quality between tools and furthermore to identify strengths and weaknesses of them. Therefore we need a quality model that allows to compare the results of various tools, since their rules differ and consequently not directly comparable to each other.

The identification of the preciseness of the results requires a classification of the findings into at least two categories, the true positives (TP) and the false positives (FP). A finding that is a TP actually is a potential flaw and detected correctly. FPs are being listed as violations against rules too, but they are detected falsely. They are no potential flaws.

According to [3] a "*confusion matrix*" helps to represent a "*binary decision problem*", where items have to be classified as positive or negative. We already explained the positives above. True negatives (TN) are items that are correctly classified as being negatives (not a potential flaw in our meaning). False negatives (FN) are incorrectly classified as being negatives, they should be positives (a potential flaw in our meaning, but not detected). Typically, there are some metrics supporting to understand the results, e.g. *True Positive Rate* or so-called *Recall* (TP/(TP+FN)), *False Positive Rate* (FP/(FP+TN)), or *Precision* (TP/(TP+FP)); Since the method only considers the classification of findings into TP or FP, all metrics including true or false negatives are not suitable at this. On this account, we define within this paper the false positive rate (FPR) as the proportion of the FP to the sum of FP and TP, or with respect to [3] as the value of 1 minus precision.

The expected result of an application of the method is a list of rated findings. On this basis, it is possible to calculate the FPR, which indicates the preciseness of static code analyzers very well.

According to [1] and [11] the FPR reaches from about 1 percent [1] to 47 percent [11] for the tool FindBugs. In [11] the authors mention a FPR of 96 percent for a tool called QJ Pro [8]. However, the FPR depends on the version of the tool (including its configuration) as well as of the analyzed source code and, as mentioned above, differs immense. Furthermore, it depends on what defines FPs. The authors are discordant to this issue, [9] also alludes to this topic. Hence, we need a method that standardizes the evaluation of the quality of tools, in particular the preciseness described by the FPR.

2. Approach / Methodology

The principal idea of our approach is to apply different static code analysis tools to a defined set of source code and to conduct the analysis on these results, with an emphasis on finding FPs. Therefore, it is necessary to define the reference source code that serves as input for all tool analysis processes as described below. In our analysis, the reference code consists of three open source projects in order to cover different programming styles and application domains. In [11] the authors mention the influence of different programming styles and experience, emphasizing the need of various source code projects. These three open source projects are: Azureus Version 2.5 which is a file sharing client; JFreeChart 1.0.2 which is a Java library to display diagrams graphically; DrJava Stable Release 20060918 which is an integrated development environment for Java [6].

The method for assessing the quality of static code analyzers splits up to five different phases. The analysis of every version of a specific static code analysis tool passes the following phases.

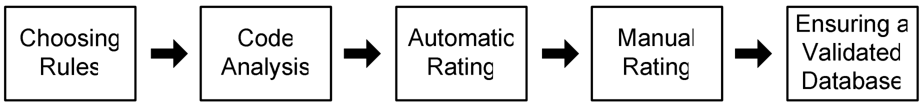


Fig. 1. Phases of the method

At the beginning of the analysis, we have to select the rules of interest. In general, all rules of the next lower version of the tool are used plus the newly added ones. If you investigate a tool for the first time, the default rule set serves as a basis, if available. If no default rule set is available, you have to select the rules manually to ensure that similar rules of already analyzed tools are used.

After defining the rules, you have to apply the static code analysis tool with the respective rule set configurations to the reference source code. After this analysis the resulting findings are converted to a homogeneous format, guaranteeing the possibility to rate them in later phases.

The data collected in the previous step will now be rated automatically. Due to the large amount of data (e.g. for PMD approximately 12.000 findings obtained by the analysis of the above-mentioned reference code), this is a necessary step. Therefore, a tool was developed (*SimpleEval*) that rates the analyzed findings using a heuristic algorithm and a validated database. It compares every finding with the already rated findings within the validated database. For this purpose, it considers the source code file as well as the line number within the file. It is configurable how close the line numbers have to be. Furthermore, it takes into account the descriptions of the violation of the comparing findings. If another version of the same tool has already been rated, a flag can be enabled that controls only to look at rated findings of the same code analysis tool. The heuristic results in a measure for the probability to be the same semantic finding. After calculating the probability for all rated findings near the unrated one, *SimpleEval* applies the rating of the finding with the highest probability to the unrated finding. Moreover, it classifies the probability in four categories as follows: A25 indicates a very low equality; A50 indicates a middle equality; A75 indicates a high equality; A100 indicates a very high equality.

The output of *SimpleEval* contains the following information for each rated finding: the name of the tool; the version of the tool; the name of the rule which is violated; the pathname as well as the filename where the finding is found in; the line number within the file the finding occurs; a flag which defines the finding as TP or FP; a flag which defines the finding being rated manually or by *SimpleEval*; a description of the violation. Another tool reads this format and enables to process, sort and search for findings very easily.

The findings that remain unrated or rated lower than A100 have to be rated manually. That can be a large amount and depending on the rule a high effort. This happens gradually for each rule that still lists unrated findings. For this purpose a sample of at least 30 percent of the unrated findings of every rule is declared to be rated manually. If one finding of this sample is a FP, all other findings of this rule need to be rated by hand too. However, if every finding of the sample is a TP, we assume all other findings of this rule to be TPs too. After finishing the rating of all remaining findings, they are classified as one of the following values: TP, FP, or NA that means that the finding cannot be rated because of a too vague description of the rule or a too difficult or time-costly rating process.

The last phase of the method ensures a valid database by checking a sample of the newly rated findings for a second time. An expert will do this. If the expert finds no wrong rated finding, all newly rated findings can be added to the valid database and subsequently be used for automatic rating with *SimpleEval*. If the expert finds a wrong rated finding, all manually rated ones have to be rated and subsequently reviewed again.

3. Results

The result of using this method is a benchmark file containing many findings rated automatically or manually. At this time it contains the results of the analysis processes of the tools PMD in version 3.7, 4.0, 4.1, and 4.2 plus CodePro AnalytiX in version 5.2 and 5.3. It currently lists about 90.000 rated findings. However, using this benchmark file allows us to gain information about the quality of several versions of tools. On this basis, we can determine the progress of a tool's quality over different versions (intra-tool comparison) as well as how good or bad is a tool's quality compared to another one (inter-tool comparison). For this, a quality model helps to determine the strengths and weaknesses of a tool at specific quality attributes. Every quality model would be suitable for this reason, but we developed a technical classification (TIC – technical issue classification) that is similar to the ISO 9126 quality model [4]. It consists of a problem-oriented classification. It emphasizes the needs of architects or technical project managers. Therefore, all rules of the tools within the benchmark file have to be assigned to quality attributes. The count of TP or FP as well as the FPR of a specific rule therefore serves as metrics within the quality model.

As already mentioned above, we used this method and tool to analyze the quality of the tools PMD and CodePro AnalytiX. Now let us take a closer look at the progress of PMD's quality.

Table 1. Overview of PMD's quality progress

Version	NA	FP	TP	FP+TP	!	FPR
PMD 3.7	644	185	12839	13024	13668	1.42%
PMD 4.0	624	188	13068	13256	13880	1.42%
PMD 4.1	636	89	12937	13026	13662	0.68%
PMD 4.2	636	87	12949	13036	13672	0.67%

Table 1 shows the number of findings classified as NA, FP or TP for every investigated version. It only considers the rules analyzed in PMD 3.7, ignoring all newly added rules in later versions. The column that gives us the most valuable information is the last one, the FPR. The FPR drops from 1.42% to 0.68% in version 4.1 because of reduced FPs. This is a sign of increasing quality. Not only the count of FP falls in version 4.1 compared to version 4.0, but also the count of TPs falls. This is a sign of decreasing quality. Altogether, it is difficult to state the development of the quality here, since it furthermore depends on the importance of the findings being lost in version 4.1. Comparing version 4.1 with 4.2 let us recognize a slight reduction of FPs combined with a small increase of TPs. This combination results in a better quality of version 4.2 compared to 4.1 that also explains the lower FPR.

This is one example of the information we can gather about a tool's quality. We can create and interpret the same table for every rule of a tool, or every quality attribute within a quality model. Furthermore, it is possible to take all findings of all rules into account.

Furthermore, an assignment of metrics to a quality model enables to analyze the distribution of the findings over several tools. Fig. 2 shows this distribution for the tools PMD and CodePro AnalytiX over nine quality attributes of TIC. As already mentioned, TIC serves as a quality model where all metrics of PMD and CodePro AnalytiX (plus metrics of other tools) are assigned to technical quality attributes.

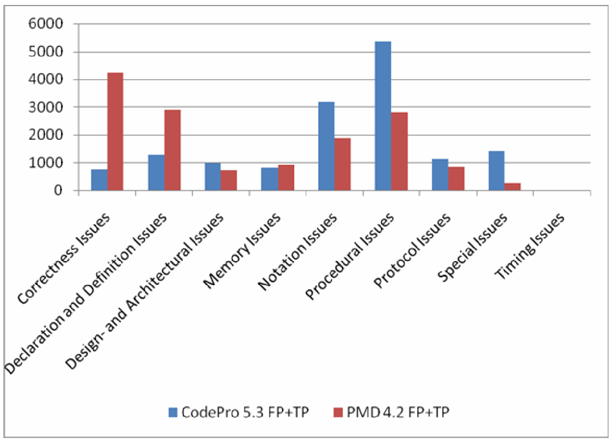


Fig. 2. Distribution of findings for PMD and CodePro AnalytiX in context of TIC factors

We can see the main focuses concerning identified positives of a tool, e.g. PMD 4.2 at the factors Correctness Issues or Declaration and Definition Issues. This does not automatically infer from that, to indicate good quality. On the other hand, CodePro AnalytiX finds a lot more violations against rules at Notation Issues, Procedural Issues or Special Issues. We will not explain the quality factors of TIC in more detail, since it is not the focus of this paper and they only help us to illustrate results of a first application of the method and the tool.

Now let us look at the tool's FPR of every quality factor in TIC, as presented in Fig. 3.

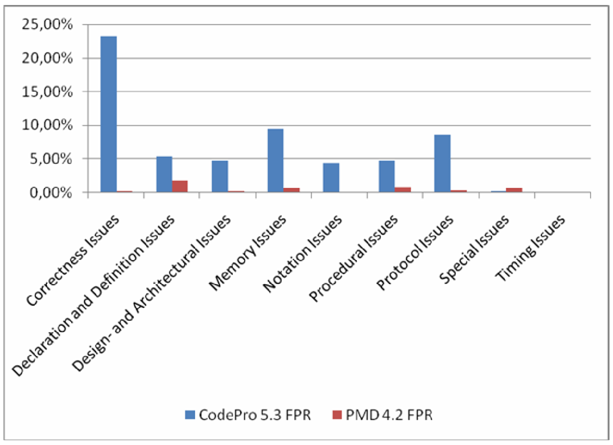


Fig. 3. FPR of every quality factor in TIC of PMD and CodePro AnalytiX

This figure shows a higher FPRs of CodePro AnalytiX in comparison to PMD, except in the quality factor Special Issues. In general, the tool with the lower FPR should be preferred to analyze source code, as it is more precise. Nevertheless, it depends on the damage that occurs if some positives remains ignored due to the fact that findings have not been identified by the tool with the lower FPR. The combination of more detected findings with a lower FPR like in Correctness Issues prioritizes PMD for usage. CodePro AnalytiX should preferably analyze Special Issues due to its lower FPR and the higher count of findings.

4. Conclusion

In order to be able to apply static code analyzers that work fine, there is a need to know how good or bad they or specific versions of them work. As chapter 3 shows, the method and tool that were implemented provide valuable results about the quality of static code analyzers. Due to these results, it is possible to compare tools or tool versions that enable to make funded purchase decisions. Therefore, statements about the preciseness of tools or several rules are possible and enable a sustainable selection of rules for the input of a quality model. Additionally, it is possible to increase the trustworthiness into tools as they can be evaluated before potential usage.

As a result of our analysis processes, we know all true positives, false positives, and false positive rates of every metric within every code analysis tool and their versions we have investigated. These results can easily be illustrated using common diagramming tools.

A validated database is essential and its quality is crucial, because it serves as basis for the automatic rating of findings. It contains no information about false negatives. False negatives are flaws which have not been found by the tool [10]. It is extremely difficult to define all flaws within the reference code that would be necessary to define the false negatives of an application of an analyzer. In an outlook, we could consider false negatives, as we compare the true positives of a tool with those of another tool or tool's version. The difference of the number of true positives is the number of false negatives. Nevertheless, this comparison does not ensure to find all false negatives (of the version with less true positives), since the comparing version does not ensure to find all positives.

We have some ideas for improvements of *SimpleEval*, in order to reduce the need for manual rating to a minimum. Firstly, the heuristic that is used to automatically rate the findings should be improved. If a tool is rated for the first time, many findings cannot be matched to already rated findings of another tool, because the descriptions of the violations differ too much. Secondly, a matching of similar rules of different tools could result in better outcomes concerning the equality of two findings. Based on this *SimpleEval* could rate more findings with the highest equality level (A100). This will effectively reduce the need for manual rating, which is very time-costly. The effort to apply the method including both, automatic rating using *SimpleEval* and manual rating, is about two man-days if an earlier version has already been analyzed. If a new tool will be analyzed the effort increases to about five to ten times.

Due to the existing large database and the existing tool support, it will be increasingly simpler to assess the quality of new versions of a tool and even new, not yet considered, tools automatically.

Literature

1. Ayewah, N., Pugh, W., Morgenthaler, J. D., Penix, J.: Evaluating Static Analysis Defect Warnings On Production Software, 2007
2. CodePro AnalytiX: Product information about CodePro AnalytiX can be obtained via http://www.instantiations.com/codepro/analytix/index.html
3. Davis, J., Goadrich, M.: The Relationship Between Precision-Recall and ROC Curves, Technical Report, 2006
4. DIN ISO 9126: Informationstechnik – Beurteilen von Softwareprodukten, Qualitätsmerkmale und Leitfaden zu deren Verwendung, 1991
5. FindBugs: Product information about FindBugs can be obtained via http://findbugs.sourceforge.net
6. Höllerer, M., Schrögenauer, J.: Systematische Evaluierung von statischen Codeanalysewerkzeugen – Konzepte und Werkzeugunterstützung. Diplomarbeit, Institut für Wirtschaftsinformatik Software Engineering, Universität Linz, 2007; In German.
7. PMD: Product information about PMD can be obtained via http://pmd.sourceforge.net
8. QJ Pro: Product information about QJ Pro can be obtained via http://qjpro.sourceforge.net
9. Rutar, N., Almazan, C., Foster, J.: A Comparison of Bug Finding Tools for Java, 2004
10. Shatkay, H., Feldman, R.: Mining the Biomedical Literature in the Genomic Era - An Overview; Journal of Computational Biology, Vol. 10, No. 6, 2003, S. 821 - 855
11. Wagner, S., Jürjens, J., Koller, C., Trischberger, P.: Comparing Bug Finding Tools with Reviews and Tests, 2005

Comparability of Assessment Results

Bhaskar Vanamali, Markus Müller

KUGLER MAAG CIE

Abstract. In recent years- the number of [ISO/IEC 15504] Assessments has increased continuously. Alone in the automotive industry - particularly in Europe – [ISO/IEC 15504] more than 300 official Assessments have been performed only by HIS members till end of 2006 (source: Spokesman of HIS group). The number of internal assessments in the automotive industry is estimated more than twice as much. Other branches are also performing [ISO/IEC 15504] Assessments. If you compare the results of successive assessments or the expected assessment result after a preliminary assessment with the “official” result what kind of delta is to be expected. Based on our experience with this kind of situations we will prepare a comprehensive study of deltas and the reasons. At the moment we have only preliminary results which we will prove as the study is continuing.

Experiment for Comparing the Automatically Assessed Source Code Quality with Experts' Opinions

Harald Gruber[1], Anja Hentschel[2], Reinhold Plösch[1]

[1] Johannes Kepler University Linz, Altenberger Straße 69,4040 Linz, Austria
{harald.gruber, reinhold.ploesch}@jku.at

[2] Siemens AG, Corporate Technology – SE 1, Otto-Hahn-Ring 6, 81739 Munich, Germany
{anja.hentschel}@siemens.com

Abstract. The evaluation of software quality is supported by numerous tools but is still an extensive task that has to be carried out manually by an expert. We present a method for an automatic assessment of source code quality by using a benchmarking-oriented approach to rate the results of static code analysis tools. Within an experiment we compared these results with the evaluations of several experts who made a ranking of the software projects regarding to their quality. As a result we can reveal that the experts' ranks strongly correlate to the ranking of our automatic assessment method. The approach is promising with the restriction that we just made statements about a quality ranking of the software projects and skipped conclusions about the absolute quality.

Keywords: Software Quality, Automatic Assessment, Static Code Analysis, Benchmarking

1 Introduction

Managing the quality of a software project is a key success factor for a software project. Numerous approaches are available to make the term quality operational in a sense that it is possible to assess the quality of a software product. One field focuses on systematically refining the term quality by means of a quality model - like the well established ISO 9126 model [1] and the successor ISO/IEC 25000 [2]. Other research groups (e.g. [3], [4] or more actual [5]) identified important metrics for measuring the quality of a software system. Finally there are static code analysis tools available that check whether a software product adheres to principal well defined coding best practices. Wide spread representatives of such tools for the Java programming language are FindBugs [6] and PMD [7].

Although considerable progress has already been made in the above research areas, the evaluation of the quality of a specific software project is still an extensive task that is usually done manually by an expert (see e.g. proposed methods in [8] and [9]).

Alternatively, an automated quality assessment approach has been developed in scope of the QBench project [10] and is described in detail in [11]. The QBench approach uses a set of 52 rules emphasizing object-oriented issues and rate them by comparing the number of rule violations with benchmark data of about 100 software projects. Then the quality level of the software is automatically calculated using a defined but inflexible approach. One problem of this approach is that the automatic assessment method is so strict that hardly any project goes beyond the lowest quality level. Details on our experiences with the QBench approach can be found in [12].

Based on this experience we developed an automatic assessment method that is based on the ideas of QBench. As one major difference we are not tied to a fixed set of metrics and base our benchmarks on the metrics provided by any static code analysis tools. Furthermore we propose calculation variants for aggregating the results of different rules or metrics to a final quality statement. The details of the assessment method are explained in section 2. Technically, we have a more flexible approach that leads to more differentiated quality results when applied to different software projects. Based on our model and our calculation methods we can therefore distinguish software products with good internal quality and with bad internal quality. This is promising. Nevertheless the question remains unanswered, whether this calculation result has something to do with the "real" quality.

For this purpose we conducted an experiment where we asked experts to assess the internal quality of five different open source software projects for which we calculated quality statements with our automatic benchmark-oriented approach. The experiment took place on one day where every participant had to assess the five projects in limited time. The basic idea of this experiment is to find out, whether the benchmark-oriented assessment can keep up with a human quality expert.

Section 2 briefly introduces our automatic assessment method and section 3 explains how the setup for the experiment has been arranged and how the evaluation task of the experts has been carried out. In section 4 we present and discuss the results of our experiment and give a conclusion in section 5.

2 Automatic Assessment Method

We developed an automatic assessment method based on the ideas of the benchmarking-oriented evaluation method QBench project. According to [13] benchmarking is the process of comparing products, services and practices with the strongest competitors on the market. Learning from the best is one typical way to use benchmarking; nevertheless we use this approach for comparing a software project with other projects on the market aiming at determining the quality of the software in relation to the benchmark base.

Obviously the benchmark base, i.e., the projects contained in there have an effect on the benchmark results, as the quality of a project is always relative to the benchmark base; it makes a difference whether the benchmark base contains average quality projects or projects of extraordinary quality. We provide a flexible way to select the reference projects that shall be used for the benchmark of a project. This is one major difference from the QBench approach where this flexibility is not given, but only a fixed set of projects form the benchmark base.

Furthermore we allow a flexible use of metrics and rules and therefore tools for static code analysis. Besides SISSy [14], a tool that was implemented in the scope of the QBench project, we primarily use FindBugs and PMD for Java and PC-Lint [15] for C/C++ projects.

Typical rules these tools search for are for instance whether all String comparisons are used correctly, closely related methods for a class are implemented or simply naming conventions are obeyed.

The evaluation process in detail starts by comparing the number of rule violations of the desired rules with the respective values that are stored in the benchmark base. As it makes no sense to compare the absolute number of rule violations the results are normalized by the size of the particular project measured by Logical Lines of Code (LLOC, i.e., counting only lines that contain source code). The comparison with the benchmark is done by building a statistical distribution of the values from the reference projects for every rule. When working with quartiles the result for a single rule can be one out of six: Below or equal the benchmark minimum, between minimum and the lower quartile (1st quartile), between lower quartile and median (2nd quartile), between median and upper quartile (3rd quartile), between upper quartile and benchmark maximum or greater than the maximum. Fig. 1 illustrates a possible distribution. As there is a separate distribution available for every rule we do not have a problem that the range of values for every rule can be different.

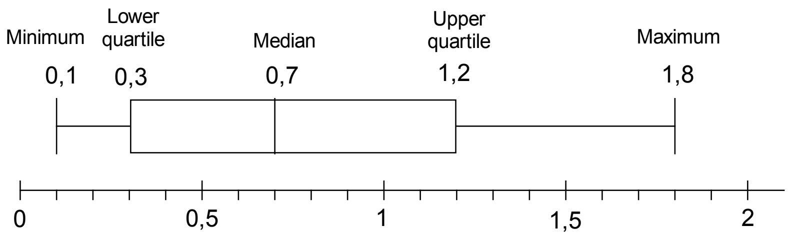


Fig. 1. Example of a quartile based distribution for a single rule calculated by the normalized values of the reference projects in the benchmark base.

We have developed several methods for calculating a grade for the overall quality or for assigning a quality level. The calculation method to use depends on the target of the assessment, for example if you want to award a kind of certificate for a software project you have to determine the conditions for certain quality levels. For our experiment, where we just want to rank projects regarding to their quality, the simple calculation of an average grade from the results of the used rules is sufficient. Fig. 2 shows the formula for calculating this grade. The formula is kept very simple and we weight every rule equally. In our opinion it would be little credible to use different weights for particular rules as we use more than 500 rules for calculating the average grade.

$$Avg = \frac{\sum_{i=0}^{i=5} NoR_{Qi} \cdot (5 - i)}{NoR_{Total}}$$

Fig. 2. Calculation of the average grade (Avg) by multiplying the number of rules (NoR) with a rating factor between 0 and 5 depending on the performance of the respective rule which is defined by the quartile achieved (Q_i). For instance the number of rules that do not exceed the minimum (Q_0) is multiplied with 5 while the number of rules that exceed the maximum (Q_5) are multiplied with 0. The sum of the rated rules is divided by the total number of rules (NoR_{Total}), so the grade can have a value between 0 and 5 where a higher value is better.

The ranking of the software projects is very simple when using the average grade. The project with the highest value will receive the 1st place while the project with the lowest value will take the last place. If two projects should get the same average grade, they will obtain the same rank.

3 Setup and Realization of the Experiment

The aim of the experiment was to determine, whether quality experts rank projects in a similar way than the automatic benchmark approach. For this purpose, experts have to assess and rank five selected software projects and their ranking results are compared to the rankings of our automatic assessment method.
The projects of the benchmark base as well as the projects to assess should be open source projects as it would be hardly possible to perform such an experiment with industrial projects. Furthermore all projects should use the same programming language to make them comparable. We decided to concentrate on the Java programming language as sufficient tools for static code analysis are available and the setup for static analysis is easier and typically more accurate than with other languages like C++.

The selection of the projects was done together with the creation of the benchmark base. The idea was to collect data from well known and wide spread Java open source projects and to choose five of them for our research. Altogether 31 projects have been analyzed, so 26 projects can be used for building the benchmark base.

The selection of the five projects for the experiment was not done randomly but by applying some criteria. There should be both small and larger projects, projects with few and projects with many rule violations and the projects should be either well-established or they should at least have a graphic user interface so that experts can also try out the applications. Finally we ended up with the five projects Checkstyle [16], JabRef [17], Log4j [18], RSSOwl [19] and TV-Browser [20] where we typically took the last stable release. The size of the projects is between 13,000 and 66,000 Logical Lines of Code (LLOC). Table 1 lists the five projects and their values of the selection criteria.

Table 1. The values of the selection criteria for the chosen projects.

Open source project (and release number)	Rank regarding rule violation/LLOC	Project size	Awareness level	GUI
Checkstyle 4.4	1	small	high	no
JabRef 2.3.1	28	medium	low	yes
Log4j 1.2.15	14	small	high	no
RSSOwl 1.2.4	9	medium	medium	yes
TV-Browser 2.3.1	11	large	medium	yes

The experts were not chosen randomly, but it was important that they have practical experience in software development, in software quality management and know the Java programming language well. The nine experts that have participated in the experiment work either in a software development company, are software consultants or academics working at an institute related to software engineering.

The assessment by the experts was arranged on one day where they got one project after the other and had to make statements about the quality attributes *analyzability*, *craftsmanship*, *reliability* and *structuredness*. The experts didn't know in advance which projects they would have to evaluate and they had to do their work simultaneously and independently of each other.

The way for assessing the projects was left up to the experts. Due to the short time - a project had to be assessed in a time between 45 and 60 minutes - it was clear that no deep analysis would be possible. So if an expert wanted to base his evaluation on the results of static code analysis tools he was free to use one. In fact two of the experts made use of this possibility. Although one can argue that these evaluations must be similar to the automatic assessment, there is a difference whether to judge a project automatically by the findings or to go manually through findings and analyze the possible problems found by the tools.

Another item is the order the projects are evaluated. The order can influence the experts as this kind of evaluation (short time, five projects at one day) was new for all of them. So for the first project they had to get accustomed to this situation and for the following projects there were already comparisons to the foregoing projects possible. Furthermore the last project had to be evaluated late in the afternoon were the concentration could be lower than in the morning. To prevent that the experts' assessments do not differ among each other just due to these facts, all experts had to evaluate the projects in the same sequence. Moreover the experts got for instance more time to evaluate the first project for compensating the acclimatization period. At the end of the day every expert had to rank the five projects accordingly to his opinion of the projects' software quality. For this final ranking of the projects the evaluation order should be negligible.

4 Results of the Experiment

The main concern of the experiment was to check whether the automatic ranking correlates with the experts' opinions about the quality of the investigated open source projects. Further evaluations regarding single quality attributes stand on a too weak statistical base to make accurate statements.

The automatic ranking was calculated by using the results from the three static code analysis tools FindBugs, PMD and SISSy with their default rule sets. We calculated an average grade for every project as it is explained in section 2 and ranked the five projects respectively. Table 2 lists the ranking when calculating an average grade by using the results of all three static code analysis tools together and the ranking with the results for every tool separately. The rank with all tools is not a combination of the separate tool ranks but an independent calculated rank using the formula for calculating the average grade with all 562 rules of the three tools. For the following analysis we use only the ranking that has been calculated with the results of all tools. Thus the ranking is 1st place for Checkstyle, 2nd place for RSSOwl, 3rd place for Log4j, 4th place for TV-Browser and the last place for JabRef.

Table 2. Automatic ranking of the projects with all tools and with each tool separately.

Open source project (and release number)	All tools	FindBugs	PMD	SISSy
Checkstyle 4.4	**1**	1	1	2
JabRef 2.3.1	**5**	5	5	5
Log4j 1.2.15	**3**	3	3	4
RSSOwl 1.2.4	**2**	2	2	1
TV-Browser 2.3.1	**4**	4	3	3

The automatic ranking was compared with the rankings of the experts. The nine experts ranked the projects unequally. This shows that the selection of the projects looks good as the source code quality of the projects is not obvious. Nevertheless the correlation coefficient between the automatic evaluation and the experts' ones lies - besides one outlier - between 0.6 and 0.9 and is hence statistically significant for more than the half of the experts. Additionally we summarized the several evaluation results of the experts to a kind of *reference expert* by taking either the ranks of the majority or the median ranks. These *reference expert* results were as well correlated. Table 3 shows the particular correlations between the experts (and the *reference experts*) with the automatic ranking by using either Pearson's or Kendall's correlation coefficient.

Table 3. Correlations between experts' rankings and the automatic ranking.

Experts	Automatic ranking (all tools)	
	Pearson correlation	Kendall correlation
Expert 1	0.900	0.800
Expert 2	0.700	0.600
Expert 3	0.900	0.800
Expert 4	0.700	0.600
Expert 5	0.756	0.671
Expert 6	0.600	0.400
Expert 7	0.900	0.800

Expert 8	0.900	0.800
Expert 9	0.866	0.738
Reference expert majority	*0.900*	0.800
Reference expert median	*0.962*	0.949

The reference experts correlate equal or even better than any expert alone. This can be interpreted in that way, that the automatic ranking achieved by tools' results establishes a kind of common denominator of the experts.

5 Threats to Validity

The experiment for comparing an automatic quality assessment with the opinion of experts has some limitations. The selection of the studied projects is a critical issue and another selection could lead to a different result. Furthermore our pool of possible projects was restricted to open source software; general conclusions about commercial software would only be possible with great care.

We let the experts evaluate only five projects because a higher number of projects would overburden every expert and it would be more difficult to get an adequate number of persons for such an experiment. On the other hand this small number doesn't allow us to perform detailed statistical analyses.

We tried to get a wide spectrum of experts with different background, but of course a bigger number of experts would give the experiment a more robust base. But it was more important for us to get evaluators with experience than to increase the number of participants.

Finally it is not possible to make appropriate statements about particular quality attributes as the statistical base for these correlations is too weak and there are too many sources of irritation for such analyses.

6 Conclusions and Further Works

The automatic ranking of the five open source projects correlates well with the rankings given by several experts that assessed the quality of these projects on one day. So the automatic assessment method seems to be a good way to get a quick and objective insight into the overall source code quality of a software project with low effort. But of course one has to consider that the presented result is just a ranking of projects and we make no statement about the absolute quality with our automatic assessment method. Furthermore the time the experts have spent on the evaluation was limited and we don't know if a detailed review would have lead to different results.

To get more information about the significance of the automatic assessment it would be interesting to perform further experiments with other experts, other projects and also other programming languages. Moreover we will try to correlate the results with other data like bug reports or results of unit testing.

7 Acknowledgments

We want to give a big sincere thank to Wolfgang Beer, Klaus-Peter Berg, Gernot Binder, Christian Danner, Andreas Hangler, Markus Maier, Herbert Prähofer, Andreas Pramböck, Gerhard Scheutz, Stefan Schiffer, and Josef Templ who were the participants of the experiment that spent their valuable time for assessing the selected software projects by hand and made this study possible at all.

References

1. ISO/IEC 9126-1:2001: Software engineering - Product quality - Part 1: Quality model (2001)
2. ISO/IEC 25000: Software engineering - Software product Quality Requirements and Evaluation (SQuaRE) - Guide to SQuaRE (2005)
3. McCabe, T. J.: A Complexity Measure. In: IEEE Transactions on Software Engineering, vol. 2, no. 4, pp. 308--320 (1976)
4. Chidamber, S. R.; Kemerer, C.F.: A Metrics Suite for Object Oriented Design. In: IEEE Transactions on Software Engineering, vol. 20, no. 6, pp. 476--493 (1994)
5. Gui, G., Scott, P. D.: New Coupling and Cohesion Metrics for Evaluation of Software Component Reusability, International Conference for Young Computer Scientists, IEEE Computer Society, pp. 1181--1186 (2008)
6. Product information about FindBugs can be obtained via http://findbugs.sourceforge.net
7. Product information about PMD can be obtained via http://pmd.sourceforge.net
8. Martin, A.E., Shafer, L.H.: Providing a Framework for Effective Software Quality Assessment - A First Step In Automating Assessments. In: Proceedings of the first Annual Software Engineering & Economics Conference, McLean (1996)
9. Plösch, R., Gruber, H., Hentschel, A., Körner, Ch., Pomberger, G., Schiffer, S., Saft, M., Storck, S.: The EMISQ Method - Expert Based Evaluation of Internal Software Quality. In: Proceedings of 3rd IEEE Systems and Software Week, Baltimore, IEEE Computer Society Press (2007)
10. Information about the QBench Approach can be obtained via http://www.qbench.de
11. Simon, F., Seng, O., Mohaupt, T.: Code-Quality-Management - Technische Qualität industrieller Softwaresysteme transparent und vergleichbar gemacht. dpunkt Verlag, Heidelberg (2006)
12. Gruber, H., Körner, Ch., Plösch, R., Pomberger, G., Schiffer, S.: Benchmakring-oriented Analysis of Source Code Quality - Experiences with the QBench Approach. In: Proceedings of the IASTED International Conference on Software Engineering, Innsbruck, Austria, (2008)
13. Camp, R.: Benchmarking – The Search for Industry Best Practices that Lead to Superior Performance, New York (1994)
14. Product information about SISSy can be obtained via http://sissy.fzi.de
15. Product information about PC-Lint can be obtained via http://www.gimpel.com
16. The open source project Checkstyle is available at http://checkstyle.sourceforge.net
17. The open source project JabRef is available at http://jabref.sourceforge.net
18. The open source project Log4j is available at http://logging.apache.org/log4j
19. The open source project RSSOwl is available at http://www.rssowl.org
20. The open source project TV-Browser is available at http://www.tvbrowser.org