Software Quality Assessment in Practice: A Hypothesis-driven Framework

Markus Schnappinger, Mohd Hafeez Osman, Alexander Pretschner
Technical University of Munich
Munich, Germany
{schnappi,osmanm,pretschn}@in.tum.de

Markus Pizka, Arnaud Fietzke
itestra GmbH
Munich, Germany
{pizka,fietzke}@itestra.de

ABSTRACT

Software quality models describe decompositions of quality characteristics. However, in practice, there is a gap between quality models, quality measurements, and quality assessment activities. As a first step of bridging the gap, this paper presents a novel and structured framework to perform quality assessments. Together with our industrial partner, we applied this framework in two case studies and present our lessons learned. Among others, we found that results from automated tools can be misleading. Manual inspections still need to be conducted to find hidden quality issues, and concrete evidence of quality violations needs to be collected to convince the stakeholders.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**;

KEYWORDS

Software Quality, Quality Assessment, Software Maintenance

ACM Reference Format:
Markus Schnappinger, Mohd Hafeez Osman, Alexander Pretschner, Markus Pizka, Arnaud Fietzke. 2018. Software Quality Assessment in Practice: A Hypothesis-driven Framework. In *ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (ESEM '18), October 11–12, 2018, Oulu, Finland*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3239235.3268922

1 INTRODUCTION

Explicit attention to characteristics of software quality can lead to significant savings in software life-cycle costs [1]. Since the 1970s, several quality models have been introduced, such as Boehm's model [1], FURPS model [11], ISO/IEC 9126 [12] and ISO/IEC 25010 [13]. Those models mostly describe the characteristics of software quality. To this end, the measurement and the assessment activities to evaluate these quality characteristics are still not precisely defined. In the 1980s, Garvin [9] pointed out that the notion of quality is subject to the perspective taken. Product quality is addressed by the ISO 9126 standard describing technical aspects of quality, while ISO 25010 also discusses the quality in use perspective. Trying to establish efficient quality control mechanisms, tools such as ConQAT[1], PMD[2], Findbugs[3], Sonarqube[4], Teamscale[5] and various coding style checkers were developed. As Koc et al. [14] point out, these tools typically detect many false positives. This claim is supported by Voas' finding, that static metrics can only measure the structure of software, but not the quality of the behavior [21]. Another approach to software quality are activity-based models, as proposed by Deissenboeck et al. in [5] and Broy et al. in [2]. Based on these models, Wagner et al. [22] tried to close the gap between abstract quality models and quality assessments. As shown in literature, software does indeed age [17] and software maintenance costs increase over time [7]. Hence, Pizka and Panas established the concept of software health checks in [18]. Still, these quality reviews need to be convincing to overcome developers' intrinsic resistance against assessments [19].

Goal. Motivated by the above context, we aim at closing the gap between abstract quality characteristics, quality assessment activities, and concrete aspects to look into when trying to assess these abstract qualities. This would provide a richer basis for judging the quality itself than purely metric-based approaches. Because this is a work in progress, we exemplify the idea with one quality, namely maintainability, and a few insights we got from our case studies. Our goal is to make software quality assessments systematic and structured. The approach described in this paper follows a structured hypothesis-driven framework. It was formulated based on 15 years of our industrial partner's practice, our observations, and literature review. This framework includes the following activities: comprehension of software artefacts, building hypotheses, performing analyses, collecting evidence and devising conclusions. Together with our industrial partner, we have used this framework to conduct quality assessments for two real-world systems. These systems are from a well-known domain (financial), are medium to large in size, and provide reasonable artefacts (i.e. source code, documentation).

Contribution. The contributions of this paper are: (i) we introduce a structured software quality assessment framework; (ii) our activities in performing quality assessments of real-world software are briefly explained; and (iii) finally, we identify limitations of tool-based software quality assessments.

[1]http://www.conqat.org
[2]http://pmd.sourceforge.net
[3]http://findbugs.sourceforge.net
[4]https://www.sonarqube.org
[5]https://www.teamscale.com

2 RELATED WORK

In terms of research related to quality frameworks, Cavano and McCall [3] provided a framework to give insights into software quality. Their research indicated that such frameworks can be useful to identify quality problems early in the development and result in large cost savings. Meanwhile, Lochmann [15] proposed a quality meta-model describing the structure of quality models. This meta-model can be used to define a hierarchy of properties, that influence the quality of a software system.

In the following, we present work that reports observations from quality assessment in practice. Gousios et al. [10] analyzed open source systems and used the publicly available product and process data to determine their quality. They enhanced the static code analysis by also taking bug management data and other data from the version control systems into account. Stürmer and Pohlheim [20] mainly focussed on the quality of model-based software projects. They assessed the systems not only with respect to quality criteria form the standards, but with respect to the success of quality-related operations like reviews and testing. Combining classical viewpoints of quality with those of the EU in [23], Wolski et al. evaluated projects of an EU-funded program. They regarded both process and product-related measurements to compare the quality of projects internally and externally. Yoo and Yoon [25] focussed on dependability as the most important quality aspect. They assessed software quality through test cases and applied formal checking to verify that the dependability requirements are implemented in the system.

Meanwhile, Fagerholm et al. [8], discuss a model for continuous experimentation. Their iterative approach features phases for building hypotheses, performing experiments, and learning from the results before creating another hypothesis.

From our initial observation, there is not much work that discusses quality assessment in practice and explains in detail the process, tools and measurements used. In this paper, we present practical quality assessment in detail and introduce a tailorable framework to analyze the quality of a system.

3 CASE STUDIES

3.1 Case Study A

In this case study, an offer management system used by an insurance company was evaluated. The owner of the system wanted to know the current quality status and asked for recommended short-term actions that can be deduced from these insights. According to the owner, the time to market provided by the system is non-satisfactory. The introduction of new tariffs or products requires high effort and is time-consuming. The system has been maintained for more than 15 years and was technically developed based on Model Driven Development (MDD). The system maintenance is outsourced to an offshore provider with approximately 50 developers working on it. Table 1 shows more detailed information about the case study A.

3.2 Case Study B

This system is an offer calculation system for brokers developed and used by an insurance company. It has been developed for the past 15 years by a group of 3 developers using Delphi. Now, the management considers to replace it with a state-of-the-art technology. Therefore, the owner asked for a structured review of the technical state of the program, a comparison with similar systems and a draft of recommended actions for mid-term improvement. Table 1 summarizes the case study B.

Table 1: Case Study A and B

Items	Case Study A	Case Study B
Domain	*Financial (Insurance)*	*Financial (Insurance)*
Purpose	*Offer Management System*	*Offer Calculation System for Brokers*
Dev. Language	*Java*	*Delphi*
Release (year)	*2000*	*2000*
Development	*Outsourced*	*In-house*
Maintenance	*Outsourced*	*In-house*
Provided Artefacts	*Source Code, Architecture Document, Data Model, User Manual*	*Source Code, Architecture Document, Data Model*
Size	*Medium-Large (1.8M SLOC)*	*Medium (486k SLOC)*

4 QUALITY ASSESSMENT FRAMEWORK

Combining our industrial partner's 15 years of experience and our own insights, we composed a framework that describes a structured, activity-based assessment process for software systems. Such an assessment is different from a Modern Code Review and can rather be seen as a general post-release software audit. Its purpose is to evaluate a given software system in order to find potential quality defects and recommend actions to improve the quality of the system. This is sometimes called software health check [18]. Before the assessment process started, there were several preliminary activities conducted, such as a kickoff meeting, a brief demonstration of the system and handing over the artefacts. The quality assessment framework is illustrated in Figure 1.

4.1 Input

As shown in Figure 1, we grouped the input information into three categories. One category captures system and project artefacts. Typical examples of system artefacts are source code, data model, documentation, and other material needed to reproduce the technical project. For context comprehension, we collect context information that is not stated in the software artefacts as well. Among these are current issues of the system, observed quality in use deficits, the expectations of the stakeholders, the background and history of the system, and more. The third category of input information is the system owner's motivation for the assessment, i.e. known problems, questions to be answered, and the trigger for the assessment.

4.2 Process

The quality assessment process consists of three phases: overview, building hypotheses, and analysis.

Overview: This phase aims at acquiring a high-level understanding of the software by mapping the functionality of the system and the software structure. The functionality of the system is derived from the documentation that was provided by the system owner. Also, we conducted an informal interview with a quality consultant that is familiar with the software domain to capture several common functionalities of such systems.

Building Hypotheses: Figure 1 visualizes that the phase of building up hypotheses consists of two aspects: *Initial hypothesis* and *working hypothesis*. The *initial hypothesis* is an assumption about the quality of the system and its problems that is purely based on intuition and not on evidence from any detailed analysis. Quality consultants implicitly build a first hypothesis based on the information from the kickoff meeting, the problems motivating the assessment, and their knowledge about common quality defects. In contrast, the *working hypothesis* characterizes an assumption about the quality of the system, that is at least partially based on evidence but still needs to be confirmed through further analysis. *Working hypotheses* represent refinements of *initial hypotheses*.

Analysis: The actual analysis is guided by the hypotheses and is divided into three activities.

- *Selection*: This task relates quality models to the assessment. The quality consultants specify the system parts to be analyzed and choose what facts are to be observed in these files. This selection is driven by a quality model, in our case the fact-and-activity matrix presented in [2]. In this model, cost-intensive maintenance activities are correlated with observable facts within a software system. Sometimes a lack of information becomes obvious at this stage, forcing the quality consultants to inquire additional input from the system owner.
- *Assessment*: The presented framework allows quality consultants to use any quality assessment method. Depending on the quality attributes and the available information, they can apply either automated, semi-automated or manual methods. In this task, the system is evaluated with respect to the specified criteria.
- *Evidence Collection*: In order to provide valuable feedback to the owner, it is important to augment the observations and conclusions with tangible evidence. We select evidence that represents average findings as well as very severe instances. Code fragments, for example, can illustrate performance issues, or concrete instances of misleading comments provide proof for statements about the quality of comments. Even measurements from automated tools can be useful evidence if they are put in the right context and are supported by concrete examples. As the evidence provides indication either in favor of the current hypothesis or against it, the working hypothesis is adapted according to the new insights and the iterative analysis is continued.

4.3 Output

After several rounds of iteration, the quality consultants come to a conclusion. This conclusion is combined with the context information about the system to identify possible root causes for the observed quality defects. Adequate actions to improve the quality are deduced and proposed to the owner. This last step has a more strategic nature and is therefore omitted in this paper, as we focus on the technical review. In addition to a final presentation, intermediate presentations were delivered to keep the owner up-to-date and provide an opportunity to discuss the next steps.

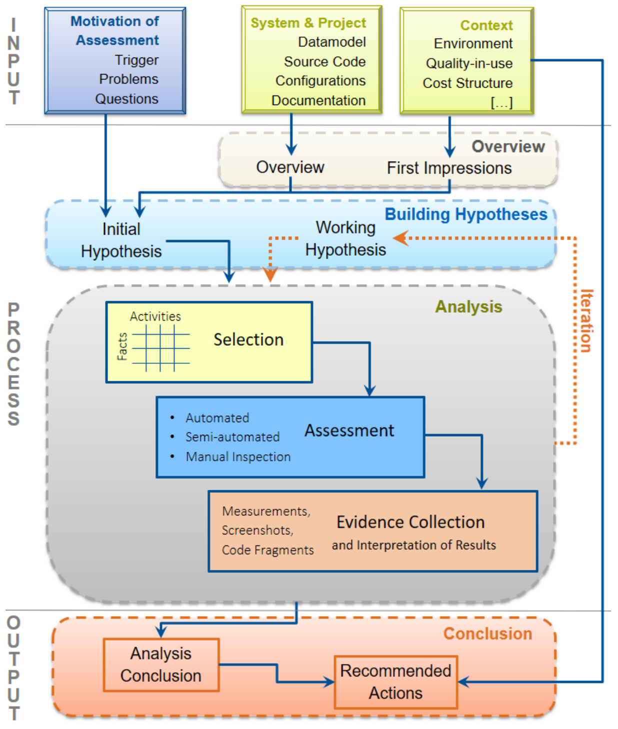

Figure 1: Quality Assessment Framework

5 THE QUALITY ASSESSMENT

5.1 Case Study A

Input: The list of inputs for case study A can be found in Table 1. Overall, the size of the provided system files exceeded 2 GB and included more than 10,000 files labeled as documentation. The source code was mostly written in Java and consisted of approx. 16,000 Java files. There was only little information provided about current costs, open requirements and the bug history. The data model was provided as a graphical representation spread over several documents. As described in Section 3, we also knew that the software faces a time-to-market problem, due to time-consuming implementation changes needed to introduce new products.

Process: The results and findings of this phase are the following:

Overview: Since the software documentation only showed parts of the software design, we rediscovered the low-level software structure by constructing UML diagrams from the source code using the tool Enterprise Architect CASE [6]. From these diagrams, it is almost explicitly shown that the modules of this software are structured according to the system functionality. Hence, the mapping between functionality and structure is straightforward.

Building Hypothesis: From a product quality viewpoint, the software faces a modifiability issue that decreases its maintainability. Therefore, we formulated the initial hypotheses based on the maintainability matrix from [6] and our own experience. For example, redundancy affects the modifiability of a system. The quality of code

[6]http://www.sparxsystems.com/products/ea/

comments and identifier naming influences the maintainability as well. Hence, the initial hypotheses pertain to these aspects.
Analysis: The activities in this stage were performed based on the formulated hypotheses. Table 2 summarizes the initial hypotheses that were refined over time, the assessment activities performed in each iteration step as well as findings and evidence. Since the length limitation of the paper, we only explain five hypotheses.
Output: In the context of this case study, the quality of a system describes the degree to which the fitness-for-purpose is reached while the running costs of the system are kept minimal. Running costs can be maintenance costs or operating costs. The maintenance effort is determined by maintenance activities, that can be related to observable facts inside the product [2] whereas operating costs are negatively influenced by ineffective implementations. Hence, a software system is considered *good* if the software fulfills its purpose at minimal running costs. Source code is considered *good* if the way it is written does not increase maintenance costs compared to other solutions and there is no reasonably achievable way to make it more efficient.

Table 2 only illustrates a small part of the findings. More activities had to be performed to evaluate the complete system. At the end of the assessment, we suggested two alternatives to the system owner: (i) Develop a new system that incorporates new technology for better scalability; or (ii) Buy and customize an available standard product. This conclusion was mainly influenced by the comparison of the maintenance costs and the perceived cost for replacing the existing software.

5.2 Case Study B

Input: The list of inputs for case study B is denoted in Table 1. Even though the artefacts provided were fewer than in case study A, the artefacts were up to date and easy to walk through. The assessment was supposed to answer whether the system is technically sustainable and suitable for future requirements.
Process: The results and findings of this step are the following:
Overview: In this case study, we refer to the software architecture and the source code to get a high level understanding of the system's functionality and program structure. Although the software architecture was presented in a simple way, it was sufficient to understand the program structure.
Building Hypotheses: The system has been maintained by the same 3 in-house developers for 15 years. Due to a high risk of lost knowledge after staff turn-overs, we concentrated on maintainability and understandability of the source code. Therefore, the initial hypotheses focus, for example, on code comments and bad coding practices.
Analysis: In Table 2, we illustrate 2 examples of hypotheses, the activities conducted in each iteration, and the corresponding findings and evidence.
Output: The information illustrated in Table 2 is only a small part of the findings from this assessment. The quality issue regarding the maintainability of the system is not severe and only improvement actions with minimal effort are recommended. However, this system is suggested to be replaced for strategic reasons. In order to be ahead of their competitors, the owner needs to acquire a new state-of-the-art software that offers more functionalities and has a better technical sustainability.

6 DISCUSSION AND LESSONS LEARNED

The presented framework is regarded as a general description. Due to the project-specific nature of quality assessments, the framework is kept tailorable and flexible. Nevertheless, we are working on concrete guidelines on how to apply each step of the framework in detail and will provide them at a later stage of our research. The remainder of this section explains our lessons learned after assessing the quality of the study objects and discusses possible threats to validity.

6.1 Assessing Software Structure

When we conducted the software structure review, we observed several metrics. However, even though some of those metrics indicate an issue of the software, we experienced differently when searching for the concrete evidence. In many cases, the metrics were just misleading. For example, the tool identifies classes with more than 60 attributes and operations as 'god classes'. This term describes a class that tends to centralize the intelligence of a system or has multiple responsibilities [16]. Looking at the concrete instances, we found that the calculation of attributes and operations also includes constants, getters, and setters. In our point of view, those do not add any significant responsibilities to a class. Thus, the tool led to many false positives which had to be excluded.

6.2 Interpreting Redundancy Ratios

In our study, we observed very high cloning ratios. But when examining concrete instances, we found that lots of the files accountable for this number were actually generated files. Usually, quality analyses omit generated files, but in case study A, many generated sources were not labeled as such and were mixed with hand-written code. After identifying and removing these sources, the observed redundancy was significantly lower. Another example from our case study are files that define constants. Being almost identical from a structural point of view, these files were categorized as clones although their content is unrelated.

6.3 Providing Evidence

Measurements are meaningless unless they are backed up by tangible, convincing and representative evidence. Even though several measurements can provide some sort of benchmark, providing evidence is necessary to prove the reliability of the measurement. In addition, evidence is helpful to demonstrate the negative consequences of found quality defects. For example, developers are more likely to accept cloning as a dangerous practice if they are presented concrete code snippets together with the observed statistics.

6.4 Analysing Identifiers

Browsing source code and performing program comprehension, we experienced the importance of identifiers for this task. We can confirm the claims from Deissenboeck and Pizka in [4], who showed that a concise and consistent naming of attributes is crucial for software readability. From our observations, we can support that finding and add another dimension to it: All names should be taken from the same language. In our case study, we found identifiers stemmed from English, French, and German. Since we performed this task manually, we did not collect statistic information about

Table 2: Initial hypotheses, assessment activities, evidence and findings for Case Study A and B

Case Study A

Item	Description
Hypothesis 1	**Poor program structure obstructs the program comprehension.**
Selection	Source Code, Architecture Document
Assessment	(i) Reverse engineer source code into UML diagrams; (ii) Extract object-oriented design metrics using SDMetrics [24]; (iii) Evaluate the size and coupling. metrics.
Evidence	Structural information and KPIs about the system; Examples of nested packages and operations with many parameters.
Key Findings	181 out of 1027 packages (17.62%) were found to have a high nesting value ($\geqslant$ 6 levels). Out of these 16126 classes, 633 classes (3.93%) show symptoms of god classes. 9.6% of 65522 operations have a long parameter list ($\geqslant$ 6 parameters).
Hypothesis 2	**The software suffers from code cloning that decreases the software maintainability.**
Selection	Source Code
Assessment	(i) Calculate cloning ratio using ConQat; (ii) Identify generated code; (iii) Re-calculate cloning ratio; (iv) Evaluate the code cloning measures.
Evidence	Duplication KPIs from ConQAT that describe the code clones and ratio; Examples of duplicated code that illustrate a copy and paste policy in a specific package; Examples of generated code.
Key Findings	With 34.4%, the cloning ratio is considered high; The amount of generated files was surprisingly high.
Hypothesis 3	**The identifier naming convention is either not existing or not maintained which affects the program comprehension.**
Selection	Source Code
Assessment	(i) Randomly select source files from key modules; (ii) Observe identifier naming convention patterns; (iii) Discover identifier naming convention violations.
Evidence	Examples of inconsistent namings; Examples of different languages used for identifiers.
Key Findings	A global naming convention could not be identified; Multiple languages were used to name identifiers.
Hypothesis 4	**The poor quality of code comments affects the program comprehension.**
Selection	Source Code
Assessment	(i) Examine all comments in the key modules and randomly selected other modules; (ii) Discover issues related to those comments.
Evidence	Examples of commented-out code; Examples of confusing comments; Examples of comments with mixed languages.
Key Findings	A lot of source code was found in comments; Comments pointing to known problems (i.e. "Todo" and "Fixme" annotations) were found in productive code; Comments are sometimes ambiguous or mingle multiple languages.
Hypothesis 5	**There are issues in the data model that lead to a decreased modifiability of the system.**
Selection	Data Model
Assessment	(i) Acquire a high-level comprehension of the data model; (ii) Discover the high-level data model concept or pattern; (iii) Investigate low-level issues of the data model such as duplication, missing normalization or inconsistencies.
Evidence	Examples of repeating attributes; Instances of missing normalization or standardization.
Key Findings	Attributes appear in several tables. Partially, these attributes have inconsistent field types. The schema was not normalized.

Case Study B

Item	Description
Hypothesis 1	**Bad coding practices such as duplicating code and hardcoding values decrease the quality of the system.**
Selection	Source Code
Assessment	(i) Browse source code from key modules and identify hardcoded identifiers; (ii) Discover code duplication using ConQAT; (iii) Evaluate the findings.
Evidence	Examples of duplicated code and its location; Examples of hardcode values.
Key Findings	Code duplication is a practice in this project. Hardcoded values like paths were found in the source code.
Hypothesis 2	**The poor quality of code comments affects the program comprehension.**
Selection	Source Code
Assessment	(i) Examine all comments in key modules and randomly selected other modules; (ii) Discover issues related to those comments.
Evidence	Examples of commented-out code; Examples of unuseful comments.
Key Findings	A lot of source code was commented out; Comments were used for informal discussions or as optical delimiters, provided no additional information or used ambiguous language.

the naming anomalies. The list of found anomalies was evidence enough to explain the issue to the product owner.

6.5 Evaluating the Data Model

In case study A, the data model was evaluated manually since only data model diagrams were provided as images. In case study B, the data model was provided as an SQL script. Browsing the models, we uncovered several anomalies. For example, it was impossible to store more than one telephone number per person. In addition, there was no strategy to manage history data - a task crucial for insurance companies. Our lesson learned here is that the data model is a valuable source of information. It is created to support the current functionality and therefore a lack of flexibility in the model can point to problems when introducing new functionalities.

6.6 Assessing Code Comments

Comments are a useful possibility to augment source code with additional information. However, we experienced that comments are often misused. The use of comments to discuss open issues instead of using an issue tracking tool indicates space for improvement on the process level. The same point can be made about "Todo" and "Fixme" tags in productive systems. Also, we saw a lot of code being commented out instead of removed. This code is likely to distract maintainers and therefore increases the maintenance effort.

6.7 Threats to Validity

The internal validity of our case study can be threatened by the small group of quality consultants (2) from just one company and the small number of researchers from university (2) performing the assessment. In addition, it has to be mentioned that both examined systems were taken from the insurance domain, which can be seen as a threat to external validity. During the analysis, all analysts actively tried to avoid any bias and keep an open mindset. But as the systems were selected for a quality assessment by the owner, the quality consultants knew there was a certain likelihood to find quality deficits. Due to the motivation of the performed assessments, both case studies focused on maintainability.

7 CONCLUSION

This paper aims to relate abstract quality models to concrete quality assessments. We present our tailorable framework to perform structured quality assessments. Applying it to real-world systems in two case studies with our industrial partner, we can conclude that blindly using automated assessment tools is not sufficient, though we can still use their output. Other lessons learned are, e.g., that (i) evidence collection is crucial not only to convince the system owner, but also the software developer, and (ii) design and maintainability of the data model may reflect the flexibility of a system.

This is a part of our early work to bridge the gap between quality characteristics, measurements, and assessments. We see several ways to improve this work such as (i) refining the framework and applying it to other systems from other domains; (ii) further research on convincing evidence from the perspective of various stakeholders; (iii) automatization of the activities in this approach and (iv) building a taxonomy that shows the relation between quality characteristics, quality measures, and assessment activities.

ACKNOWLEDGMENTS

The authors would like to thank itestra GmbH for their participation in this research cooperation. In particular, we are very grateful for their commitment to sharing their expertise with us and the opportunity to present their industrial practice in this paper.

REFERENCES

[1] B.W. Boehm, J.R. Brown, and M. Lipow. 1976. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*. IEEE Computer Society Press, 592–605.

[2] M. Broy, F. Deissenboeck, and M. Pizka. 2006. Demystifying maintainability. In *Proceedings of the 2006 international workshop on Software quality*. ACM, 21–26.

[3] J.P. Cavano and J.A. McCall. 1978. A framework for the measurement of software quality. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 7. 133–139.

[4] F. Deissenboeck and M. Pizka. 2006. Concise and consistent naming. *Software Quality Journal* 14, 3 (2006), 261–282.

[5] F. Deissenboeck, S. Wagner, M. Pizka, S. Teuchert, and J-F Girard. 2007. An activity-based quality model for maintainability. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*. IEEE, 184–193.

[6] F Deissenboeck, S Wagner, M Pizka, S Teuchert, and J-F Girard. 2007. An activity-based quality model for maintainability. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*. IEEE, 184–193.

[7] S.G Eick, T.L. Graves, A.F. Karr, J.S. Marron, and A. Mockus. 2001. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering* 27, 1 (2001), 1–12.

[8] F Fagerholm, A S Guinea, H Mäenpää, and J Münch. 2017. The RIGHT model for continuous experimentation. *Journal of Systems and Software* 123 (2017), 292–305.

[9] D. A. Garvin. 1984. What does - Product Quality - really mean. *Sloan management review* 25 (1984).

[10] G. Gousios, V. Karakoidas, K. Stroggylos, P. Louridas, V. Vlachos, and D. Spinellis. 2007. Software Quality Assessment of Open Source. In *Current Trends in Informatics: 11th Panhellenic Conference on Informatics, PCI 2007*. New Technologies Publications, 303–315.

[11] Robert B. Grady and Deborah L. Caswell. 1987. *Software Metrics: Establishing a Company-wide Program*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[12] International Standard Organization (ISO). 2001. International Standard ISO/IEC 9126, Information technology - Product Quality - Part1: Quality Model.

[13] ISO/IEC. 2010. *ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models*. Technical Report.

[14] U. Koc, P. Saadatpanah, J.S. Foster, and A.A. Porter. 2017. Learning a classifier for false positive error reports emitted by static code analysis tools. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. ACM, 35–42.

[15] K. Lochmann. 2014. *Defining and Assessing Software Quality by Quality Models*. Ph.D. Dissertation. Technische Universität München.

[16] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjøberg. 2010. Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. In *2010 IEEE International Conference on Software Maintenance*. 1–10.

[17] D.L. Parnas. 1994. Software aging. In *Proceedings of the 16th international conference on Software engineering*. IEEE Computer Society Press, 279–287.

[18] M. Pizka and T. Panas. 2009. Establishing economic effectiveness through software health-management. In *1st International Workshop on Software Health Management, Pasadena*.

[19] J. Streit and M. Pizka. 2011. Why software quality improvement fails (and how to succeed nevertheless). In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 726–735.

[20] I. Stürmer and H. Pohlheim. 2012. Model quality assessment in practice: How to measure and assess the quality of software models during the embedded software development process. *Embedded Real Time Software and Systems* (2012).

[21] J. Voas. 1997. Can clean pipes produce dirty water? *IEEE Software* 4 (1997), 93–95.

[22] S. Wagner, K. Lochmann, L. Heinemann, M Kläs, A. Trendowicz, R. Plösch, A. Seidl, A. Goeb, and J. Streit. 2012. The quamoco product quality modelling and assessment approach. In *Proceedings of the 34th international conference on software engineering*. IEEE Press, 1133–1142.

[23] M. Wolski, B. Walter, S. Kupiński, and J. Chojnacki. 2018. Software quality model for a research-driven organization-An experience report. *Journal of Software: Evolution and Process* 30, 5 (2018).

[24] J. Wüst. 2018. SDMetrics. http://www.sdmetrics.com/.

[25] J. Yoo and S. Yoon. 2013. SQAF-DS: A Software Quality Assessment Framework for Dependable Systems. In *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*. IEEE, 724–725.