Journal of Software Engineering and Applications, 2018, 11, 139-152
http://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

Framework for Observing the Maintenance Needs, Runtime Metrics and the Overall Quality-in-Use

Timo Hynninen[1], Jussi Kasurinen[2], Ossi Taipale[1]

[1]School of Business and Management, Lappeenranta University of Technology, Lappeenranta, Finland
[2]South-Eastern Finland University of Applied Sciences, Kotka, Finland
Email: timo.hynninen@lut.fi, jussi.kasurinen@xamk.fi, ossi.taipale@lut.fi

How to cite this paper: Hynninen, T., Kasurinen, J. and Taipale, O. (2018) Framework for Observing the Maintenance Needs, Runtime Metrics and the Overall Quality-in-Use. *Journal of Software Engineering and Applications*, **11**, 139-152.
https://doi.org/10.4236/jsea.2018.114009

Received: January 8, 2018
Accepted: March 26, 2018
Published: March 29, 2018

Abstract

The post-release maintenance is usually the most expensive phase in the software product lifecycle from the first design concepts to the end of product support. To reduce the costs related to post-release maintenance, we propose a run-time framework for measuring software quality characteristics applying the ISO/IEC 25000 software quality and software quality in use models as the starting point. Measurement probes are linked into the software during the development phase and used to collect quality information during the run time. As a proof-of-concept, we implemented measurements in an open-source software project to demonstrate the utility of the framework. As a result, this paper presents a framework for collecting runtime metrics and measuring software quality-in-use with a systematic interface. Additionally, examples of measurement scenarios are presented.

Keywords

Software Maintenance, Software Life-Cycle, Measurement, Test Metrics, Maintenance Costs

1. Introduction

During the software lifecycle, the maintenance of the software is usually the biggest overall expense, totaling even up to 90 percent of all life cycle costs [1]. Knowing this, it is rather surprising, that the software development processes do not focus more on the maintenance phase. Instead development processes focus to enhance and offer product quality and quality-in-use improvements within the development and quality assurance steps. For example, the Scrum software

process model which is favored in many SME organizations [2], does not take into account any activities which happen before or after the active sprints, even though majority of the software related costs are not realized within this period. This issue is glaring for example in the game software development, where the current business models such as live-ops or any other free-2-play model [3], mean that basically all profits are generated during the maintenance period, not at the commitment to develop software or after delivery.

Some activity models, such as continuous delivery (CD) [4] or DevOps [5] promote more thorough integration of maintenance activities into the development activities, but the runtime monitoring and control of the quality characteristics supporting maintenance are not included. Actions such as the delivery of hotfixes, patches or customer-tailored features are part of the continuous release cycle, where development and maintenance are concurrent activities, with the development phase being one iteration ahead of the maintenance phase. However, the general infrastructure in this area is not very systematically studied. In more abstract terms, technical evaluation of the software quality is not very straightforward, since the maintenance issues and quality assurance needs are usually related to the preferred quality: Usually quality assurance during maintenance assesses, if the software system delivers the expected features or services, and achieves the necessary quality requirements. However, there are several different types of quality involved [6], and if we consider quality models such as defined by the ISO/IEC 25000-family [7], there are tens of different measurements and methods to assess the quality and quality-in-use from different perspectives.

Many existing software measurement frameworks are influenced by the ISO/IEC quality models. For example, the software maintainability measurements developed by Motogna *et al.* [8], the performance measurement framework for cloud computing by Bautista *et al.* [9], or the framework for evaluating the effect of coding practices to software maintainability by Hegedus [10]. However, previous research has been limited to cover only parts of quality models, concentrating around specific quality characteristics such as maintainability or performance efficiency. There is a need for further work with a general measurement framework, which aims to incorporate the characteristics of a software quality model to a software system during run time.

The aim of this research is to study the different methods of reducing the costs of the maintenance by directly lowering the amount of work required for the maintenance by predicting and identifying the changes in the quality characteristics. Changes in the quality characteristics serve as an early warning system of the problematic components and software failures. More specifically, we concentrate on developing a library of software measurement probes using the ISO/IEC 25000 standard series as a starting point. From our prior study [11], applying the ISO/IEC 25000 standard, we understood that the quality model is understandable enough to warrant application in the industry. The actual research questions are: "What kind of technical infrastructure would enable

identification of on-line quality characteristics and thereby maintenance issues?" and "How to incorporate a software quality model into a library of run-time metrics?"

To answer these research questions, our approach was to define a framework and implement the framework in a system to collect and monitor run-time data from an open-source application. In addition, the collected data is visualized with a separate analysis tool to monitor trends and changes between the different versions of the system and to assess, for example, resource usage for the customer environments.

In summary, this paper presents a framework for run-time software measurement. The framework consists of two different types of metrics: direct metrics, which can be recorded from a system at run time by incorporating measurement probes into the software during development; and indirect metrics, which need to be derived from the direct metrics and the knowledge of the software engineer or software specification. The framework aims to be general to warrant use in different applications but at the same time loose enough to allow developers to derive application-specific measurement.

Rest of the paper is structured as follows: In Section 2, related work is introduced; Section 3 presents the research methodology; The main contribution of this paper, the measurement framework and our proof-of-concept project are introduced in Section 4; Discussion and conclusions are given in Sections 5 and 6 respectively.

2. Related Research

Measuring a software system with different kinds of tools and metrics is not a novel idea. There are several different kinds of definitions for the use of metrics (for example Honglei *et al.* [12]) and different quality models for selecting what to test (for example Herzig *et al.* [13]), or how to select test cases (For example Fontana & Zanoni [14], Schrettner *et al.* [15], Kasurinen *et al.* [16]). These all are serious concerns, since the test cases and in general ensuring the system testability can cause one third or even half of the workload in software development life-cycle. This is mostly due to the need to capture not only the normal usage but also extraordinary uses of the system [17].

The very definition of what product quality or quality-in-use actually means is also a concern. There are several definitions such as value-based or manufacturing-based quality [6], depending on the viewpoint or the relevant stakeholders. The users have very different views on what is software or product quality, when compared to some other quality definition, like the production quality. The customers may not care at all on how low percentage of the products are faulty or how high-quality the building components are, if the product is badly designed, overpriced against its competing products or simply feels cheap or low-grade product.

As stated, the assessment of quality relies on several measurable metrics and

models, which capture the different definitions of quality. Especially for software products and their quality, there are some different models such as CISQ [18] or ISO/IEC 9126 [19], which share a number of common features. For example, in both these models the problematic aspect of defining what product quality actually is, is solved by defining a number of characteristics such as reliability or security. These characteristics in combination assess if not all, then at least most of the different aspects of software quality, and they can be selected on case-by-case basis depending on what aspects are relevant.

The ISO/IEC 9126 model is probably the most applied standardized model but it is not the most current or extensive standard in existence. The ISO/IEC 25000 Software Quality Requirements and Evaluation (SQUARE) model [7] in its core is the upgraded version of the ISO/IEC 9126 model, with the added definitions for quality in software product, and a separate model for software quality-in-use. Overall, the objective of the ISO/IEC 25000 standard family is to clarify the requirements, which should be identified to assess the software quality and ensure the success on the evaluation and reaching of the set quality objectives. Overall, the models cover 5 characteristics with 11 sub-characteristics for the quality-in-use, and 8 characteristics with 31 sub-characteristics for software product quality. These models and their characteristics are summarized in **Figure 1** and **Figure 2**.

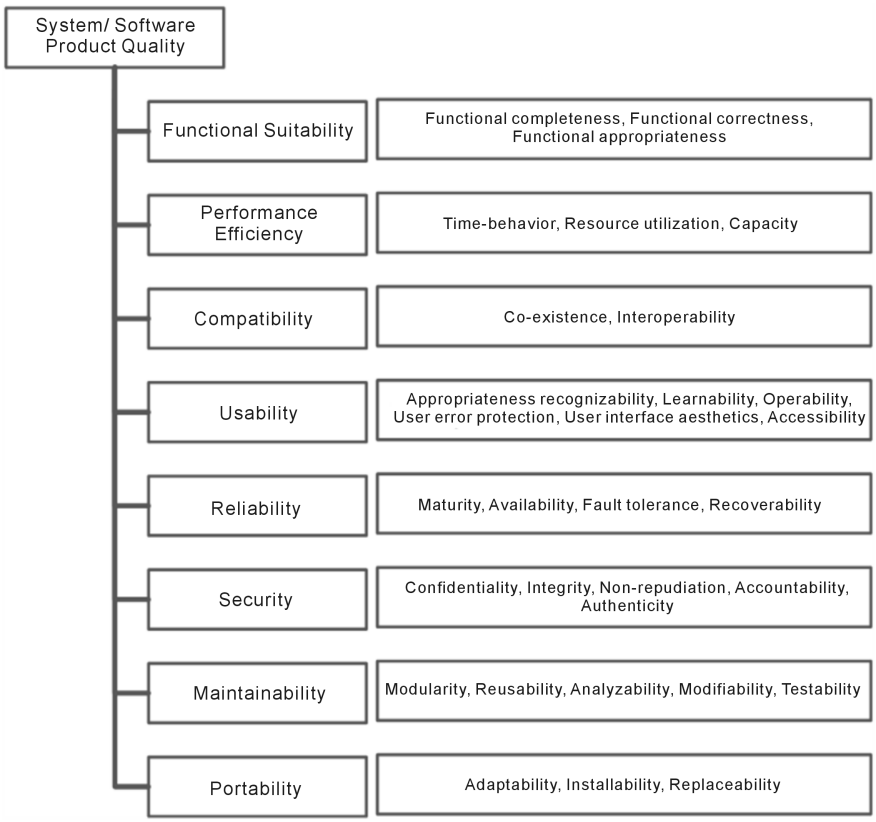

Figure 1. The ISO/IEC 25010 software product quality model [7], characteristics on left, and the subcharacteristics on the right.

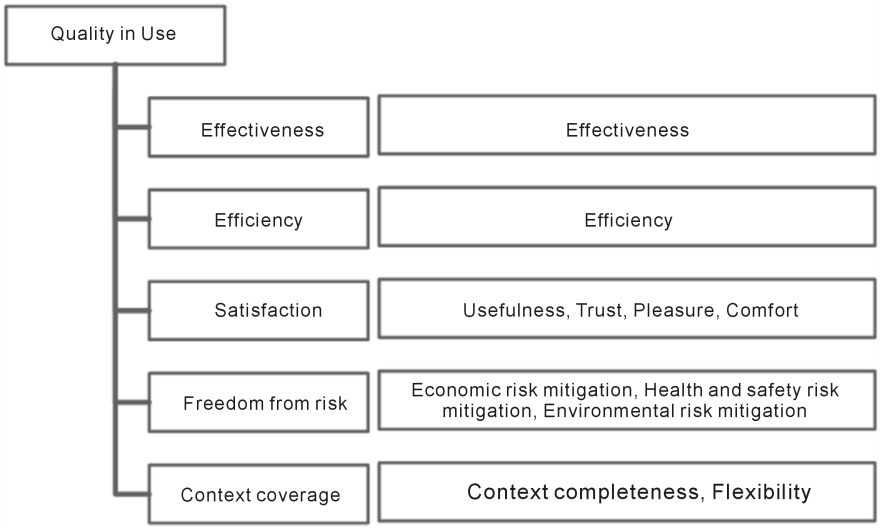

Figure 2. ISO/IEC 25010 software quality-in-use model [7].

The characteristics of the product quality model focus on the technical aspects of the software, although there are also defined sub-characteristics for more human-centric aspects such as learnability. For all of the defined sub-characteristics, the ISO/IEC 25000 standard series also defines a number of measurement techniques and metrics, which can be used to assess the quality of that particular characteristic. For example, with the testability sub-characteristics there are defined measurements for test function completeness, autonomous testability and test restartability. Similarly, confidentiality is measured with access controllability, data encryption correctness and with the strength of the cryptographic algorithms. Additionally, all of the measurements are formatted to a model, in which the result provides a clear indicator, usually percentage, of positive outcomes versus negative outcomes. Technically, this could enable the software systems to be comparable against each other, and more importantly, allow formal measurement of every different characteristic and their sub-characteristic. Similar approach is also applied in the "Quality in use"-model, which focuses on the clients and customers.

The quality-in-use-model focuses on the client side usage and on the delivered user experience. The model follows the same principle as the product quality model, dividing the model into a set of characteristics and their sub-characteristics. Unlike the product quality model, several sub-characteristics such as trust or pleasure use measurements based on the psychometric scales which are defined by a questionnaire. However, each main characteristic have at least one aspect, which can be measured through the use of software.

These models have been studied also in the other research works. For example Motogna *et al.* [8] have been studying the maintainability-characteristic of the ISO/IEC 25010 model, since in the software life cycle model maintenance has significant effect on the software costs. Their study investigates the maintenance sub-characteristics in detail, and proposes a set of metrics, which could be

applied in the assessment of the software maintainability, and provide evidence that the model is a feasible starting point for a quality assessment system. In more general studies, for example Goues and Weimer [20] have observed that the amount of needed test cases in the maintenance can be reduced almost by 55 percent, if the system is designed to include formalized method of collecting quality assurance-related metrics. A similar approach was used in a research project documented by Black [21], where a set of explicit data sources was designed to ensure that the quality assurance criterions were met in each incremental development cycle, since there were no realistic resources to do complete regression test cycle with each test case of the software during each software development cycle.

In more practical terms, Lincke *et al.* [22] have studied the different quality models and their applicability in real-life software development projects. Their study suggests, that while the models are able to implicate the quality of the software measured to some degree, the different models provide different results and the models in general are not comparable nor compatible. The same project could yield completely opposite results between two different quality models, if the selected models and applied metrics are not carefully designed and meaningful. Similar observations have been made also by Darcy and Kemerer [23], who discuss the generally applicable measurements and notify that there are only handful of universal metrics. Their studies indicate that for example in the object-oriented programming languages, concepts such amount of cohesion and coupling between the objects are the most useful metrics to assess the product quality and maintenance.

Rompaey *et al.* [17] also state that one aspect of quality, code quality, especially the concept of code smell could be transferred to the quality assurance of unit testing. Their definition of the SSVT-test cycle (set-up, stimulate, verify, tear down) could be useful in the assessment of system maintainability, test automation coverage and additional aspects such as explicitness of the system and traceability of the encountered malfunctions.

3. Research Process

During the study we constructed a framework for quality measurement and monitoring. The measurement and monitoring system aimed specifically for software maintenance using a multi-discipline approach. First, we conducted a literature review that covered, for example, software maintenance, quality assurance and software measurement methods. The review was used to identify existing solutions and proposed methods to tackle the issues raised by our research questions. In short, software quality related to the maintainability of a system is often evaluated by analyzing code quality or complexity and run-time approaches are used less often.

In addition to the literature review, we conducted a survey on the applied testing and quality assurance practices in the industry. One of the key observations

was that the use of standards and formal process models seems to have declined during the last eight years across different domains in our sample of software organizations [25]. This observation affected our approach towards an on-line measurement framework applying an international standard.

In order to realize the framework we followed the process described in the ISO 15939 (Systems and software engineering—Measurement process) [24]. Our framework covers the first two activities of the ISO 15939 model: establishing measurement commitment and planning the measurement process. The other activities recommended by the ISO 15939 model, performing measurements and evaluating the measurements, are realized as a small-scale proof-of-concept system. In the proof-of-concept, we implemented the metrics as a measurement library in an open-source application. **Figure 3** depicts the measurement framework and proof-of-concept implementations.

For each of the different sub-characteristics the ISO/IEC 25000 standard defines a set metrics or measurements, which can be applied in the assessment. For example, in the performance efficiency characteristic the sub-characteristic time-behavior is defined as follows: "The degree to which the response and processing times and throughput rates of a product or system, when performing its functions, meet requirements". To assess quality of this characteristic, the system has to be able to measure and record the response and processing times. Another example could be the compatibility-interoperability characteristic, which is defined as "degree to which two or more systems, products or components can exchange information and use the information that has been exchanged". This characteristic demands a measurement or metric to assess object interface similarities, usage of data storages and the amount of errors caused by the faulty simultaneous operations. This approach was used to establish measurements for sub-characteristics of the ISO/IEC 25000 model. Measurements were either direct measurements such as with the performance efficiency, or indicative measurements, which were used to collect information related to the characteristic.

4. Framework for Collecting and Monitoring Quality Characteristics

The concept system and the proposed testing and maintenance framework is

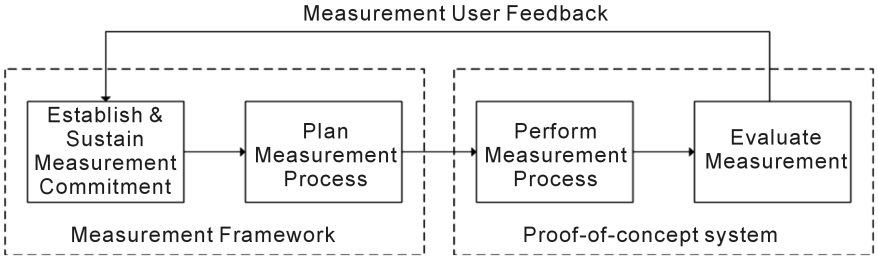

Figure 3. The measurement framework and proof-of-concept system (Modified from ISO/IEC 15939 Systems and software engineering—Measurement process model [24]).

based on two separate components, which complement each other: the metrics library, developed as a proof-of-concept IDE plugin for NetBeans [26], and the analyzer front-end, which visualizes the collected metrics.

The IDE plugin includes a shorthand and the code generators for the different types of measurement functions included in the library. The measurement functions collect data into a log file with session-relevant information, which enables the analysis tool to calculate the results and maintenance information. The objective of the IDE plugin is to offer practical tools for testers and software developers to measure and collect relevant data to satisfy their needs to verify or to validate their product, or to assess the feasibility and stability of their latest release.

The metrics library consists of different testing methods. These methods are collected from previous experiences and research work with the software industry, from different models, for example, Swebok [27], Test Process Improvement (TPI) model [28] and Test Maturity Model integration (TMMi) model [29]. The objective of the library is to offer a wide list of different testing techniques and tools, and recommend at least one feasible approach for evaluating any ISO/IEC 25000 family model characteristic.

The target IDE and the used programming language Java were both selected to represent a well-known, platform-independent development environment. The developed measurements were then incorporated to a test project, which in our case was an experimental version of the Violet UML editor [30]. The experimental version had all of the measurements implemented, so that the system would provide real session data for the analysis tool to calculate.

Table 1 presents the measurements using the quality characteristics collection and monitoring framework. The measurements are categorized as either direct or indirect: Direct measures consist of runtime events which are used to calculate descriptive statistics; Indirect measures are derived from the direct measures, and their implementation requires additional expert information from the developer or the designer. For example, Maintainability is an indirect measure based on both runtime and static analysis, whereas Mean time between failures is a direct measurement.

The analysis tool gives longitudinal observations for the product maintenance and the reveals production issues. The tool is used to analyze the existing log-files, assess quality characteristics and provide a visualization snapshot of the current state of the system along with a view into the changes of key values between the software versions. The objective of this quality characteristics collection and monitoring framework system is to provide robust and transferable metrics, which can be used to assess the "wellbeing" of the system, and provide systematic and relevant information from the state of the environment or success rate of the system revisions against the set targets. Especially for the maintenance, one long-term objective would be the observation of system performance or feature utilization.

Table 1. Ways to measure the different quality characteristics in the proof-of-concept environment.

ISO 25010 Characteristic (subcharacteristic)	Ways to measure in the framework	Measurement type	Implementation in the software
Functional suitability (Functional correctness, functional appropriateness)	Code coverage, user-applied action to achieve use case outcomes	Indirect	Analysis tool calculations with IDE plugin code insert
Performance efficiency (Time behavior)	Mean response time, response time adequacy, mean throughput	Direct	Analysis tool calculations with IDE plugin code insert
Compatibility (Interoperability)	External interface adequacy	Indirect	IDE plugin code insert.
Usability (Learnability)	Error messages understandability, user error recoverability	Direct	Analysis tool calculations, IDE plugin insert
Reliability (Maturity)	Mean time between failure (MTBF), Failure rate	Direct	Analysis tool calculations with IDE plugin code inserts
Security (Accountability)	System log retention	Direct/Indirect	Analysis tool calculations (log retention).
Maintainability (Analysability, Modifiablity)	System log completeness, Modification correctness	Indirect	Analysis tool calculations (errors after tailoring)
Portability (Adaptability)	Operational environment adaptability	Indirect	Analysis tool calculations (errors after tailoring)
Effectiveness	Task error intensity	Direct	Analysis tool calculations, IDE plugin code inserts
Efficiency	Task time	Direct	Analysis tool calculations, IDE plugin code inserts
Satisfaction	Feature utilization	Direct	Analysis tool calculations, IDE plugin code insert
Freedom from risk (Economic risk mitigation)	Business performance, errors with economic consequences	Indirect	Analysis tool calculations, IDE plugin code inserts
Context coverage (Flexibility)	Proficiency independence	Indirect	Analysis tool calculations, feature utilization-%

To evaluate the utility of the proposed framework, we developed use case scenarios to test the proof-of-concept system where the metrics library based on the framework was integrated to the Violet UML editor. In the scenarios we wanted to present simple maintenance metrics collected over time which would be beneficial for a developer monitoring a software system in use.

In the first scenario the proof-of-concept system is being used by multiple clients, with varying hardware and possibly different operating systems. The performance metric we decided to visualize was mean system startup time for each client. Figure 4 presents the data from our scenario with six different clients. In this example, the developer would be able to see if a patch or update causes system startup times to rise for all clients, and have an early warning for when to adjust loaded resources at startup. Alternatively, if a client files a bug report about slow system performance, the developer will be able to categorize if the problem appears locally for a single client only.

In another scenario, the metric we implemented was the usage of a new feature in the program. When software is in the maintenance phase old functionality

seldom changes, but new features may be added. In our scenario, a new feature has been added and deployed. In the scenario there is only one client but the software could just as well be deployed as a public web service or the metric could be the sum of all clients. The software developer wants to monitor how much the new feature is being utilized since it has been launched into production. In this example, the two features being compared allow the end user to access the same functionality and have the same outcome, but through a different path of navigation in the user interface. Figure 5 illustrates the comparison between the usages of the selected features, where feature utilization is plotted by day. As observable from the graph, users in this scenario have started to favor the newly deployed feature over the old one to accomplish their task.

5. Discussion and Conclusion

The objective was to integrate a software quality measurement framework into source code as a library of measurement tools. To bridge the gap between

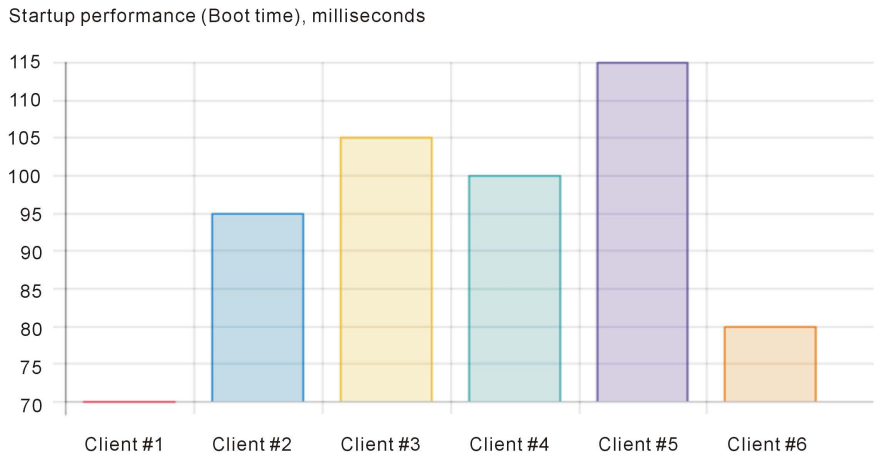

Figure 4. Example, a time-performance metric collected from six different clients in a test scenario.

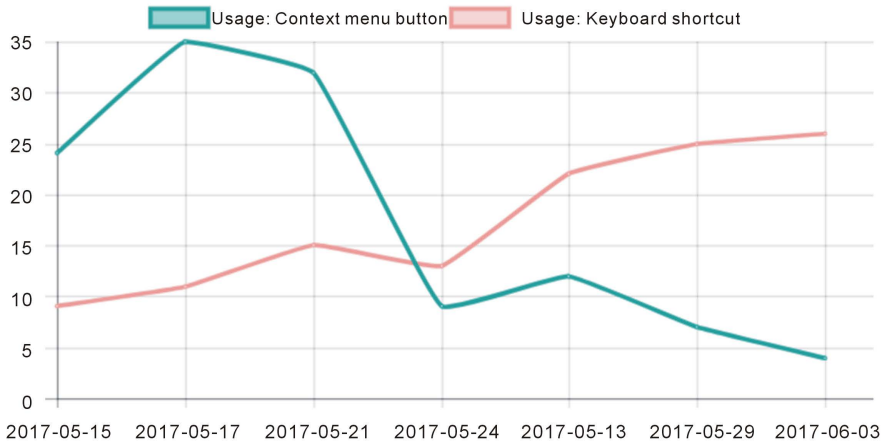

Figure 5. A feature utilization metric collected from clients in a test scenario.

established quality evaluation models and their use in practice, we used the ISO 25,000 software quality and quality in-use models as the starting point. In order to realize our goals we implemented a framework for software measurement and a proof-of-concept prototype using an open-source project, and evaluated the work using descriptive scenarios for software in the maintenance phase of its life cycle. The framework offers a step towards integrating software development and run-time quality evaluation.

According to the quality characteristics collection and monitoring framework, the ISO/IEC 25,000 quality characteristics which can be represented or measured from technical aspects of the system can be covered by the framework. The framework offers the following novel benefits:

- Development of a systematic interface for the measurement components.
- Framework that is systematic and intuitive enough to warrant ease of use without extensive training.
- Analysis tools.

Software quality related to the maintainability of a system, is often evaluated by analyzing the quality or complexity of the source code. Cyclomatic complexity [31], Halstead complexity measures [32], and C&K metrics [33] are established ways to measure code complexity. The complexity metrics are calculated directly from source code, and analysis tools often employ them. For example, Microsoft's Visual Studio includes a maintainability index indicator, which is based on both Halstead metrics and cyclomatic complexity [34]. In the academic work, RTtool is a software suite used by researchers to analyze the relative thresholds for the metrics of code quality in a software project [35]. Unfortunately, at the moment existing code complexity metrics are poorly used in the industry [36].

Model based approaches or machine learning have been identified as solutions of evaluating software and predicting defects [37]. Runtime metrics have been proposed as one method of quality evaluation [37], and they have been applied by, for example Hegedus, whose model used run-time measures together with static measures to measure testability and analyzability by using fault proneness metrics [10]. However, in general run-time metrics are rarely used in software quality and maintenance evaluation.

The limitations and validity of the presented framework warrant some discussion. First, we must acknowledge that the analysis of metrics depends on the software they are used with. Not all quality characteristics are interesting in all software applications. The analysis is affected by the application context, and therefore the normalization of metrics varies case by case.

This work begun by using the ISO/IEC 25,000 software quality and software quality-in-use models as the starting point. In the presented framework, we have covered examples of quality characteristics of the models. The limitations are related to quality-in-use characteristics that have an inherent subjective nature. For example, it is difficult to quantify user trust, pleasure or comfort through

source code, but indirect run-time measurements may give useful information. Quality characteristics like freedom from risk or security can only partly be covered.

Additionally, the utilization of the framework requires effort from the developer. Probes must be fitted directly to source code, as the framework is intended to be used considering the domain knowledge. In our proof-of-concept library we have tried to minimize the required manual programming work required by exposing ready-to-use API's to the developer.

6. Conclusions

The objective of this paper was to study how the amount of maintenance effort, and thereby, cost could be reduced using a quality characteristics collection and monitoring framework. The paper presents the implementation of a framework for software measures and a proof-of-concept prototype using an open-source project. The framework provides a systematic interface, which can be used to collect runtime metrics and measure software quality-in-use.

The measurement framework and proof-of-concept project were evaluated by using descriptive scenarios for software in the maintenance phase of its life cycle. The measurement framework was implemented as a metrics library, and measurements were linked into the software as probes during development. This work maps the run-time software metrics to quality characteristics.

In future work, we are going to investigate approaches to source code modelling and defect prediction methods to automate the measurement process. In addition, the methods presented to assess the quality of the system during maintenance could also be thematically expanded to cover the software lifecycle phases of design and implementation.

Acknowledgements

This work was funded by the Technology Development center of Finland (TEKES), as part of the. Maintain project (project number 1204/31/2016).

References

[1] The Four Laws of Application, Total Cost of Ownership (2012) Gartner, Inc., Stamford, CT.

[2] Kasurinen, J., Maglyas, A. and Smolander, K. (2014) Is Requirements Engineering Useless in Game Development? In: Salinesi, C. and van de Weerd, I., Eds., *Requirements Engineering: Foundation for Software Quality*, *REFSQ* 2014, *Lecture Notes in Computer Science*, Vol. 8396, Springer, Cham, 1-16. https://doi.org/10.1007/978-3-319-05843-6_1

[3] Alha, K., Koskinen, E., Paavilainen, J., Hamari, J. and Kinnunen, J. (2014) Free-to-Play Games: Professionals' Perspectives. *Proceedings of Nordic Digra* 2014, Gotland, 29 May 2014.

[4] Humble, J. and Farley, D. (2010) Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Pearson Education. https://books.google.fi/books?id=6ADDuzere-YC

[5] Roche, J. (2013) Adopting DevOps Practices in Quality Assurance. *Communications of the ACM*, **56**, 38-43. https://doi.org/10.1145/2524713.2524721

[6] Garvin, D.A. (1984) What Does "Product Quality" Really Mean? *Sloan Management Review*, **4**, 25-43.

[7] ISO/IEC (2011) ISO/IEC 25000: Systems and Software Quality Requirements and Evaluation (SQuaRE)—Guide to SQuaRE.

[8] Motogna, S., Vescan, A., Serban, C. and Tirban, P. (2016) An Approach to Assess Maintainability Change. 2016 *IEEE International Conference on Automation, Quality and Testing, Robotics* (*AQTR*), Cluj-Napoca, 19-21 May 2016, 1-6. https://doi.org/10.1109/AQTR.2016.7501279

[9] Bautista, L., Abran, A. and April, A. (2012) Design of a Performance Measurement Framework for Cloud Computing. *Journal of Software Engineering and Applications*, **5**, 69-75. https://doi.org/10.4236/jsea.2012.52011

[10] Hegedus, P. (2013) Revealing the Effect of Coding Practices on Software Maintainability. 2013 29*th IEEE International Conference on Software Maintenance* (*ICSM*), Eindhoven, 22-28 September 2013, 578-581. https://doi.org/10.1109/ICSM.2013.99

[11] Kasurinen, J., Taipale, O., Vanhanen, J. and Smolander, K. (2012) Exploring the Perceived End-Product Quality in Software-Developing Organizations. *International Journal of Information System Modeling and Design*, **3**, 1-32. https://doi.org/10.4018/jismd.2012040101

[12] Honglei, T., Wei, S. and Yanan, Z. (2009) The Research on Software Metrics and Software Complexity Metrics. *International Forum on Computer Science-Technology and Applications*, Chongqing, 25-27 December 2009, Vol. 1, 131-136. https://doi.org/10.1109/IFCSTA.2009.39

[13] Herzig, K., Greiler, M., Czerwonka, J. and Murphy, B. (2015) The Art of Testing Less without Sacrificing Quality. *Proceedings of the* 37*th International Conference on Software Engineering*, Vol. 1, Florence, 16-24 May 2015, 483-493. https://doi.org/10.1109/ICSE.2015.66

[14] Fontana, F.A. and Zanoni, M. (2011) On Investigating Code Smells Correlations. *IEEE* 4*th International Conference on Software Testing, Verification and Validation Workshops*, Berlin, 21-25 March 2011, 474-475. https://doi.org/10.1109/ICSTW.2011.14

[15] Schrettner, L., Fülöp, L.J., Beszédes, Á., Kiss, Á. and Gyimóthy, T. (2012) Software Quality Model and Framework with Applications in Industrial Context. 16*th European Conference on Software Maintenance and Reengineering*, Szeged, 27-30 March 2012, 453-456. https://doi.org/10.1109/CSMR.2012.57

[16] Kasurinen, J., Taipale, O. and Smolander, K. (2010) Test Case Selection and Prioritization: Risk-Based or Design-Based? *Proceedings of the* 2010 *ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, Bolzano-Bozen, 16-17 September 2010, Article No. 10. https://doi.org/10.1145/1852786.1852800

[17] Rompaey, B.V., Bois, B.D., Demeyer, S. and Rieger, M. (2007) On The Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test. *IEEE Transactions on Software Engineering*, **33**, 800-817. https://doi.org/10.1109/TSE.2007.70745

[18] Consortium for IT Software Quality. http://it-cisq.org/

[19] ISO/IEC (2001) ISO/IEC 9126: Software Engineering—Product Quality.

[20] Goues, C.L. and Weimer, W. (2012) Measuring Code Quality to Improve Specifica-

tion Mining. *IEEE Transactions on Software Engineering*, **38**, 175-190. https://doi.org/10.1109/TSE.2011.5

[21] Black, P.E. (2006) Software Assurance during Maintenance. 22*nd IEEE International Conference on Software Maintenance*, Philadelphia, 24-27 September 2006, 70-72. https://doi.org/10.1109/ICSM.2006.58

[22] Lincke, R., Gutzmann, T. and Löwe, W. (2010) Software Quality Prediction Models Compared. 10*th International Conference on Quality Software*, Zhangjiajie, 14-15 July 2010, 82-91.

[23] Darcy, D.P. and Kemerer, C.F. (2005) OO Metrics in Practice. *IEEE Software*, **22**, 17-19. https://doi.org/10.1109/MS.2005.160

[24] ISO/IEC (2007) ISO/IEC 15939: Systems and Software Engineering—Measurement Process.

[25] Hynninen, T., Kasurinen, J., Knutas, A. and Taipale, O. (2017) Testing Practices in the Finnish Software Industry. *IEEE Conference on Software Engineering Education and Training*, Savannah, 7-9 November 2017.

[26] NetBeans IDE. https://netbeans.org/

[27] Bourque, P. and Fairley, R.E. (2014) Guide to the Software Engineering Body of Knowledge, Version 3.0. IEEE Computer Society, Washington DC.

[28] Koomen, T. and Pol, M. (1999) Test Process Improvement: A Practical Step-by-Step Guide to Structured Testing. Addison-Wesley Longman Publishing Co., Inc., Bostong.

[29] van Veenendaal, E. and Wells, B. (2012) Test Maturity Model Integration TMMi. Uitgeverij Tutein Nolthenius, Hertogenbosch.

[30] Horstmann, C.S. and de Pellegrin, A. Violet UML Editor. http://violet.sourceforge.net

[31] McCabe, T.J. (1976) A Complexity Measure. *IEEE Transactions on Software Engineering*, **SE-2**, 308-320. https://doi.org/10.1109/TSE.1976.233837

[32] Halstead, M.H. (1977) Elements of Software Science. Vol. 7, Elsevier, New York.

[33] Chidamber, S.R. and Kemerer, C.F. (1994) A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, **20**, 476-493. https://doi.org/10.1109/32.295895

[34] Code Analysis Team (2007) Maintainability Index Range and Meaning—Code Analysis Team Blog. https://blogs.msdn.microsoft.com/codeanalysis/2007/11/20/maintainability-index-range-and-meaning

[35] Oliveira, P., Lima, F.P., Valente, M.T. and Serebrenik, A. (2014) RTTool: A Tool for Extracting Relative Thresholds for Source Code Metrics. *IEEE International Conference on Software Maintenance and Evolution*, Victoria, 29 September-3 October 2014, 629-632. https://doi.org/10.1109/ICSME.2014.112

[36] Antinyan, V., Staron, M. and Sandberg, A. (2017) Evaluating Code Complexity Triggers, Use of Complexity Measures and the Influence of Code Complexity on Maintenance Time. *Empirical Software Engineering*, **22**, 3057-3087. https://doi.org/10.1007/s10664-017-9508-2

[37] Catal, C. (2011) Software Fault Prediction: A Literature Review and Current Trends. *Expert Systems with Applications*, **38**, 4626-4636. https://doi.org/10.1016/j.eswa.2010.10.024