EARLY WARNING SYSTEM FOR SOFTWARE QUALITY ISSUES USING MAINTENANCE METRICS

Dmitrii Savchenko[1], Timo Hynninen[2], Ossi Taipale[1], Kari Smolander[1] and Jussi Kasurinen[1]

[1] LUT University, School of Engineering Science
[2] South-Eastern Finland University of Applied Sciences
timo.hynninen@xamk.fi, kari.smolander@lut.fi, jussi.kasurinen@lut.fi
Finland

Abstract: When software systems are developed, one of the major milestones is usually the successful launch. However, in the overall life cycle models for software, this is only the first step into the expensive and lengthy phase, the maintenance. In this study, the objective is analysing how to reduce the cost of software maintenance and manage complexity. This goal is accomplished by building an analysis tool that indicates the expected amount of maintenance work based on the first observations after a new release. We demonstrate the utility of the tool by analysing three real-world software projects. Based on the analysis findings, the maintenance indicators produced by the tool match the code review and revision needs, indicating further avenues for future development.

Key words: software maintenance, early warning system, software quality, quality assurance

1. INTRODUCTION

When considering the generic software life cycle and development models [1], the software maintenance phase is usually the last or second-to-last step with a relatively small amount of new actions or activities. However, due to the rise of the software as a service distribution methods [2] and continuous delivery models [3], software maintenance phase is arguably one of the most costly phases in the lifecycle model [4, 5]. In fact, in some software industries the first launch expects the system to include only the bare essentials, and majority of the content is developed while the system itself is in 'the maintenance phase' [6].

The growth of the maintenance phase and the costs related to the software maintenance work have been explored in many studies. Obviously there is no one main reason or culprit for the trend, but a number affecting factors such as increasing complexity and integration of the systems [7], changing operation and operating

environments of the systems [8], the criticality of the systems [5], and the rise of service-oriented approach into delivering software and their functionalities[9].

Many different approaches and technologies are aimed at the reduction of software maintenance costs including, for example, service-oriented architecture (SOA) [10], different delivery models [11], development and operations (DevOps) [12], and microservice architecture [13]. Those approaches mostly focus on improving software maintainability in order to reduce maintenance costs. On the other hand, there are techniques aimed at software quality estimation focusing on maintainability [14], code metrics [15] or code smells [16] but they require an interpretation because their key measurements are not compatible between projects

The majority of maintenance work tends to be perfective or corrective [17]. However, preventative efforts with design patterns, code smell analysis or cyclomatic complexity [18] analysis can help by identifying areas, which with high probability can raise issues. To investigate this in the context of software maintenance further, we defined following research question: *How to estimate the observed quality and maintenance needs of the software using objective code metrics*?

In order to answer this question, we developed prototypes and proof-of-concept tools and measurements, and implemented the most promising candidates on a decision support system called the .Maintain tool.

The development of .Maintain was based on the measurement principles of the quality characteristics as defined in the ISO/IEC standard 25000-SQUARE quality model [19], but introduced two further steps. In the first step, measurement units called probes are integrated into the system during the development phase to assist the data collection and activity logging work when the new feature is added during the maintenance work. Secondly, every time new version of the system is deployed, the system analyses the quality outcomes from the data collected by the probes. By comparing the analysis metrics for the relative changes in key quality factors against the historical data if the analysis tools finds a quality anomality, it triggers the early warning system (EWS) and presents the conflicting change in the quality metrics to ensure that the change is acceptable, or intentional. Based on our first deployed prototype with three different commercial software projects, the basic premise of the EWS analysis tool measurements seem to correlate with the project activity logs on the selected number of quality characteristics.

2. RELATED RESEARCH

To assess the maintenance needs, one aspect of the work is to measure, understand and improve the system and process quality. In 1988, Humphrey [20] described the framework that was aimed at establishing the standards of excellence for software engineering called Capability Maturity Model (CMM). However, these large scale approaches are not necessary applicable in all types of software projects; Hynninen et al. [21] indicates a trend that software developers tend to use as much

automation as possible, especially in quality assurance (QA), and use informal processes over formal approaches and automation tools over inspections.

Another common approach to enhance maintainability and quality of the software is to apply modularity and reusability design principles on the system architecture. [22, 23]. In object-oriented programming (OOP) [24] the system is defined as set of objects, and the isolation is mostly logical, whereas in service-oriented architecture [25] (SOA) the system is defined as a set of components which communicate as dedicated web services [10].

All mentioned approaches to reducing the maintenance costs have the common idea to reduce complexity [7]. Such an approach requires measurement tools that provide feedback about the current code complexity as early as possible. Motogna et al. [26] presented the metric based on the maintainability characteristics described in ISO 25010 [19] with the study indicating that such a metric may represent the current quality of the overall project. This observation, that particular code metrics correlate with the software maintainability is also approved by Heitlager et al. [27], who defines a metric called Maintainability index which is aimed at the representation of the software maintainability as a single metric.

Overall, there is a growing body of current research on the monitoring of metrics collected from software systems and code. Cito et al. [28] developed an IDE plugin which analyses the performance of software code, helping the developer identify components which are running slow. Similarly, Winter et al. [29] proposes the concept of Monitoring-Aware IDE's, where DevOps style monitoring is integrated into the IDE. Both studies indicate that runtime monitoring as a part of software development activities can help developers identify critical components, and better understand the system behaviour. Similar demonstrations of performance monitoring have been presented in the studies by Dawes et al. [30] and Hasselbring & van Hoorn [31]

3. RESEARCH PROCESS

Code quality may be estimated in different ways, for example, by applying both static and dynamic testing. In this study, we decided to start with the Maintainability index, but focus on the change dynamics of this index instead of the absolute value, to assess if it could act as an early warning system for the maintenance. To study this, we built a prototype analysis tool following the principles of the design science research method [32, 33]. Design science study is usually understood as research that produces constructs, methods, and models, and uses two iterative approaches: building and evaluation [33]. In practice, design science may be described by the process called design cycle; our approach adopted this practice, and it is summarized in Figure 1.

We initiated the design cycle from the problem identification of software maintenance costs and aims, while the research questions formed the initial requirements for the prototype framework. We started the design with the automated

collection of data related to most of the quality-defining attributes of the quality standard ISO/IEC 25010 [18], but immediately found out that it would require inputs which are beyond the reach of a simple data collection or repository mining tools, due to no directly applicable data being recorded. Because of this, we created the concept of probes similar to ides presented in [28, 29], a set of independent modules which can be embedded to an existing source code to measure different concepts such as transaction lengths, amounts of actions the user takes to complete one action, or other such activities. We tested the first version of the probe library using a simple open source project, and after further refinement of problems and concepts, developed a number of probe sets needed to run the quality assessment tests with real software development projects. The architecture of our solution is presented in Figure 2, and the detailed developed process, the module details and first trial of the tool is reported in publication [34].

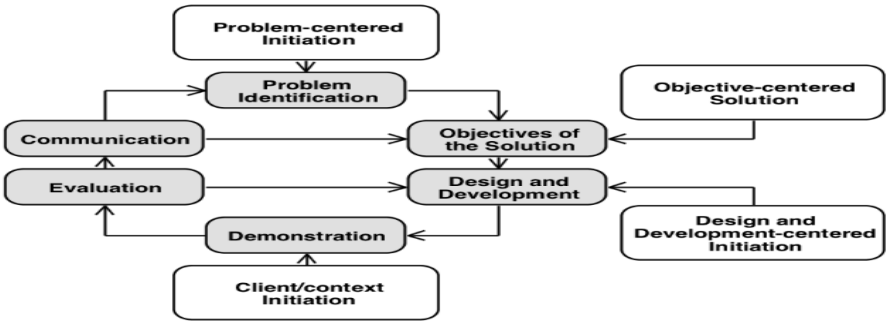

Fig. 1. Design science cycle process model

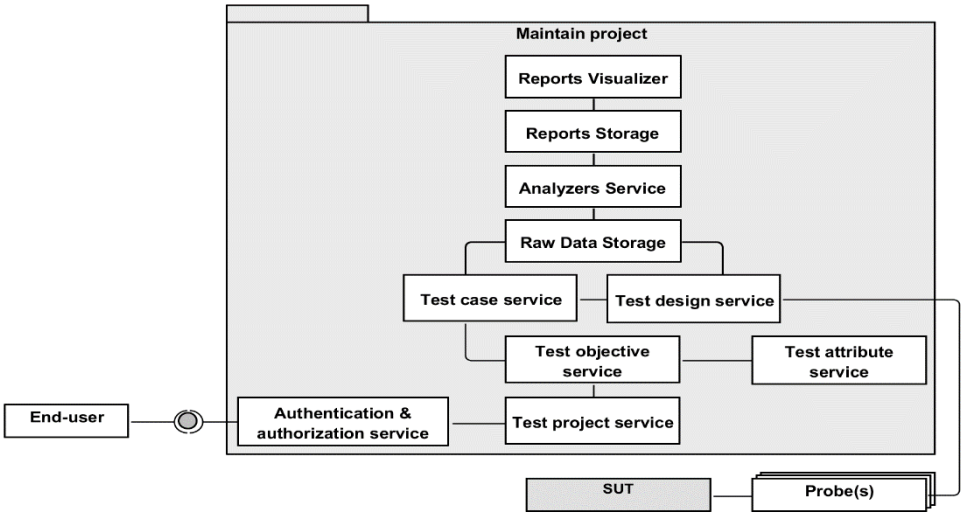

Fig. 2. Maintain-tool architecture

4. RESULTS

Design science cycle model implies both building and evaluation of the developed artifact. To evaluate the .Maintain tool, we ended up implementing the same set of code quality probes for the different programming languages and analyzed three finished real-life software projects at different stages of development pulled from their code repositories, while simultaneously collected a survey from the project managers of the said projects to provide a short summary and feedback on what their developer team was doing at the time. These results and the respective projects are presented in the following sections.

4.1. Project 1

The first project was evaluated using a proprietary application that was developed by two different teams. The application's backend was implemented using Ruby on Rails, while the frontend was developed using JavaScript and HAML notation language. The calculation metric was described using the following iterative formula, based on code smells [35] found in the analysed file:

$$qual_0 = 100$$
$$qual_i = qual_{i-1} * k \quad (1)$$

Where k is 0.99 if code smell is the warning, and 0.9 if code smell is the error. As the metrics calculations were now different and produced results in different ranges, we decided to ignore the absolute values and focus on code quality change. To extract the code quality change dynamics, we decided to use this formula:

$$trend = code\_quality - linreg(code\_quality) \quad (2)$$

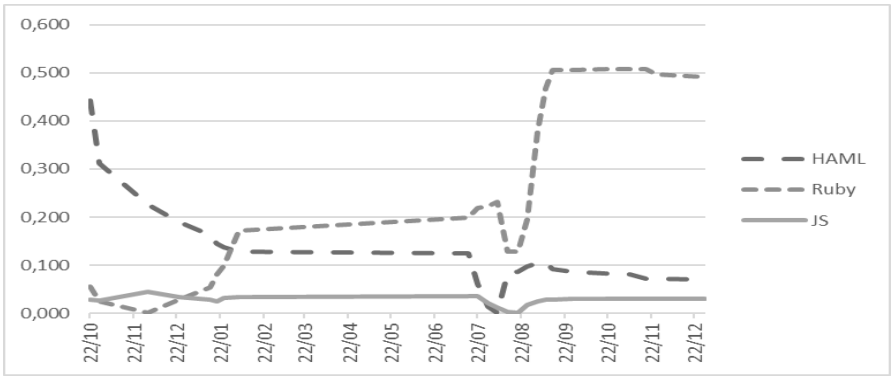

Fig. 3. Project 1 quality report

In short, code quality change was calculated as a difference between code quality measurement and the linear regression of the code quality within a given range. Figure 3 illustrates those changes for the three different parts of the project: Ruby, JavaScript, and HAML. Also, according to the feedback from the project manager, an external developer used to work with the frontend part between July 2016 and September 2016. In Figure 3, we can highlight that HAML code quality

decreased when the external developer started to work. Based on this project, we also observed that the absolute value of Maintainability index was not as critically important as the trend.

4.2. Project 2

The second pilot was performed with a project based on Node.js. The Figure 4 demonstrates the changes in the code quality over the analysed timeline, with peaks and pits being explained with a summary from the project manager questionnaire. This figure suggests that code quality is linked with the metric: maintenance index grows during the refactoring phase, and drops with the new features. Comparison between Maintainability index report and project manager feedback also revealed that Maintainability index change does not necessarily tell that something goes explicitly wrong in the project, because code quality decrease may also be linked with the project development stage, and the behaviour of lowering quality index is normal, if it goes down during activities such as new feature implementation.

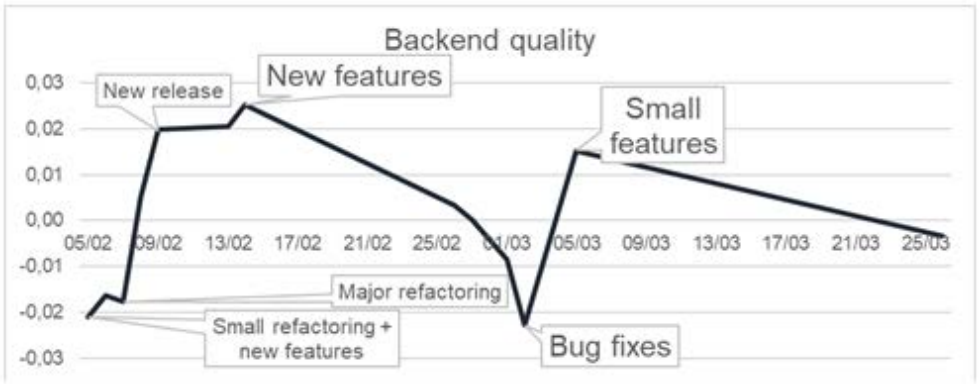

Fig. 4. Project 2 quality report with the comments from project manager report

4.3. Project 3

In the third pilot project with our maintainability observation tools, we decided to analyze a project with a timespan of one year. To get a comparison point from the available data, we decided to compare the Maintainability index analysis results with the processed ticket information from the project bug tracker. We defined and calculated the 'Bugs and features' metric using the following formula for each day:

$$y_i = y_{i-1} + N \quad (3)$$

where N is 1 if ticket type is 'Bug', or -1 if ticket type is 'Feature'. For the calculation and comparisons, the reported bugs and accepted features are included based on the ticket creation date. In any case, as observable from Figure 5, there is a correlating trend between the Bugs and features metric, and the Maintainability index change; when the bug reporting system was collecting more tickets concerning bugs and problems, the maintainability-index was going down even though the index was

based on the structural aspects of the source code, such as amount of modules, lines of code and cyclomatic complexity.

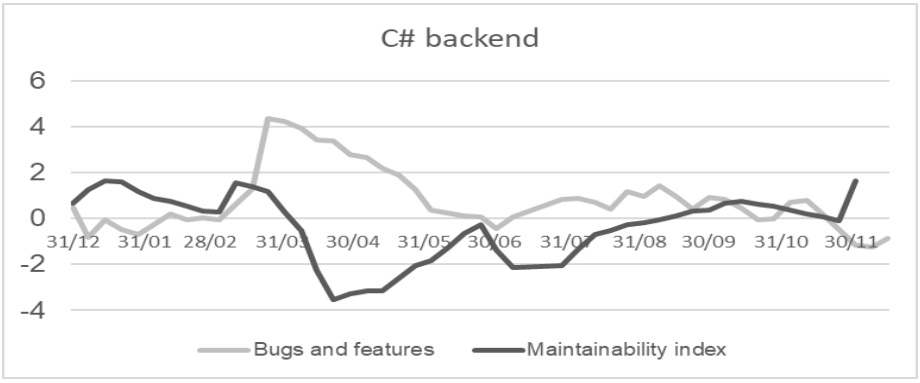

Fig. 5. Project 3 quality report with bugs and features -metric

5. DISCUSSION AND IMPLICATIONS

The modern development processes for software systems emphasize concepts such as continuous integration [11], cloud as the delivery model [36], service computing [36], collaboration between software development and maintenance [12], new integrated development environments and others. Unfortunately, the testing and deployment environments seem to be falling behind especially on the availability of generic tools due to the diversity of ecosystems. In this study, we decided to combine both static and dynamic testing to implement a tool that provides estimates of the project quality and can be integrated into the existing software development process.

Based on our three test cases with the current version of the quality assessment tools, our observations show that changes in the Maintainability index rather than the absolute value of the index gives us the opportunity to estimate the project quality. This evaluation also highlights that during the new functionality implementation, Maintainability index seems to go down, while during the refactoring or the bug fixing phase it rises. Evaluation of the Maintain project shows that by abusing this feature it might be possible to implement an early warning system that provides code quality estimation using the combination of Maintainability index and information from the issue tracking system. Such an early warning system compares the first derivative of the Maintainability index to the current project state derived from the issue tracking system. Differences between those metrics should be treated as an anomaly and reported.

Beyond our work with the maintainability estimations, the idea of finding the link between the source code metrics and project quality attracts the attention of the researchers. For example, Lewis and Henry [14] show a correlation between different code metrics and the amount of defects in the source code. However, for example Fontana [16] indicates that code smell alone can be useful for software

quality estimation and can be used as a quality metric. In any case, there are several studies [28-31] which identify strategies to assess the general software quality, with the premises similar to ours but without the basis on the quality attribute definitions from the ISO/IEC 25010-standard.

To guarantee the validity of the study we used two different approaches in the evaluation and in the methodological triangulation. Robson [37] lists three basic threats to validity in this kind of research: reactivity (the interference of the researcher's presence), researchers bias, and respondent bias and strategies that reduce their threats. To reduce these biases, we decided to perform the evaluation in two steps. As the first step of evaluation, we performed the analysis based on source code metrics during a limited time range and compared the results with the feedback from the project manager. To make the feedback more structured, we used a questionnaire and asked project managers to state their opinion on the project quality for the given time ranges. Comparison between Maintain analysis results and the special metrics derived from the issue tracking system showed the correlation, but not an exact match. This may be explained by the fact that it was not possible to link the bug or feature with the exact code change or certain commit action, because the issue tracking system used in the evaluation does not provide such information. This step of the evaluation also highlighted the fact that during the new functionality implementation, following strictly the maintainability the index will go down, while during the refactoring or bug fixing phase it will rise. Adding further analysis option with the indexes calculated from the other quality attributes of the ISO/IEC 25010 model might provide us with additional venues to collect more detailed information on why the quality is fluctuating.

6. CONCLUSIONS

In this study, we tried to answer the question - is it possible to determine the changes in the subjective software quality by objective measurements? To find the answer to this question, we decided to use a design science research approach. Design science implies the creation and evaluation of the artifact, and as the first step, we implemented the prototype of the Maintain-tool to calculate indexes based on the ISO/IEC 25010 quality attributes. Following that, we used the developed artifact to answer the research question during the evaluation phase and developed a further tool to assess the maintainability indexes. The evaluation was performed in a form of piloting within several independent companies. The evaluation showed that the Maintainability index may be used as a suitable source of the project quality estimation, but by itself it does not provide much information on why the quality is declining. In an evaluation phase with the information derived from the issue tracking system, it was possible to create an early warning system that compared code quality fluctuations to the current project stage (refactoring, new features, bug fixing) of the project. The difference between estimated the development stage and

the direction of the code quality metrics change should be reported to the project manager as a possible source of problems.

Different metrics of the source code quality has been introduced in the related studies, but the same metrics may provide different absolute values for different applications. Being aware of this, we decided to implement the Maintain as a tool focused on gathering data from different sources and analysing this data. In this study, we illustrated as a quality-in-use characteristics example, that Maintainability index can be applied to the project quality estimation and provide an early warning of issues, but at this stage the results do not provide sufficient details on what actions the maintenance team should take. Based on our observations, the early warning system is feasible to provide an alert that there might be issues within the new deployed version of the system, but automated assessment of the quality attribute changes collected from the user and system activity data to provide details on what parts of the system are failing, still need further work.

REFERENCES

[1] Lientz BP, EB, Swanson. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980.

[2] Ma, D. The business model of" software-as-a-service". In *Proc. IEEE international conference on services computing (SCC)*, July 2007, pp. 701-702.

[3] Chen, L. Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, **2** (vol. 32), 2015, pp. 50-54.

[4] Kyte A. *The Four Laws of Application Total Cost of Ownershi*p. https://www.gartner.com/doc/1972915/laws-application-total-cost-ownership. April 2012, Accessed October 15, 2018.

[5] Capgemini. *WORLD QUALITY REPORT 2016-17*. 2017.

[6] Leppänen, M., S. Mäkinen, M. Pagels, V.P. Eloranta, J. Itkonen, M.V. Mäntylä, T. Männistö. The highways and country roads to continuous deployment. *IEEE software*, **2** (vol. 32), 2015, pp. 64-72.

[7] Banker R.D., S.M. Datar, C.F. Kemere, D. Zweig. Software complexity and maintenance costs. *Communications of ACM*, 2002. doi:10.1145/163359.163375.

[8] Reisman S. Costs and Benefits of Software Engineering in Product Development Environments. In: *Cases on Strategic Information System*s. 2011.

[9] Glass R.L., R. Collard, A. Bertolino, J. Bach, C. Kaner. Software testing and industry needs. *IEEE Software*, **4** (vol. 23), 2008, pp. 55-57.

[10] OASIS. *Reference Model for Service Oriented Architecture 1.0. OASIS Standard*. Public Review Draft 2. 2006; October, pp. 1-31.

[11] Pawson R. *Continuous Delivery: Reliable Software Releases through Build, Test and Deployment Automation*, 2011. doi:10.1007/s13398-014-0173-7.2.

[12] Ebert C, G. Gallardo, J. Hernantes, N. Serrano. DevOps. *IEEE Software*, **3** (Vol. 33), 2016, pp. 94-100. doi:10.1109/MS.2016.68.

[13] Lewis J, M. Fowler. *Microservices*. http://martinfowler.com. http://martinfowler.com/articles/microservices.html, 2014.

[14] Lewis J, S. Henry. A methodology for integrating maintainability using software metrics. In *Proc. Conference of Software Maintenance*. 1989. doi:10.1109/ICSM.1989.65191.

[15] Ferreira K.A.M., M.A.S. Bigonha, R.S. Bigonha, L.F.O Mendes, H.C. Almeida. Identifying thresholds for object-oriented software metrics. *Journal of Systems and Software*. **2** (vol. 85), 2012. doi:10.1016/j.jss.2011.05.044.

[16] Fontana FA, M. Zanoni. On investigating code smells correlations. In: *Proceedings 4th IEEE International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)*, 2011. doi:10.1109/ICSTW.2011.14.

[17] ISO. *International Standard* - ISO/IEC 14764 IEEE Std 14764-2006 Software Engineering - Software Life Cycle Processes - Maintenance. ISO/IEC 14764 (2006), 2006. DOI: 10.1109/IEEESTD.2006.235774.

[18] Gill, G.K., C.F. Kemerer. Cyclomatic complexity density and software maintenance productivity. *IEEE transactions on software engineerin*g, **12** (vol. 17), 1991, pp. 1284.

[19] ISO: *ISO/IEC 25010:2011*, Systems and software engineering --Systems and software Quality Requirements and Evaluation (SQuaRE) --System and software quality models, 2011.

[20] Humphrey WS. Characterizing the Software Process: A Maturity Framework. *IEEE Software*. **2** (vol. 5), 1988, pp. 73-79. doi:10.1109/52.2014.

[21] Hynninen T, J. Kasurinen, A. Knutas, O. Taipale. Software testing: Survey of the industry practices. *Proc. Int. Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2018. doi:10.23919/MIPRO.2018.8400261.

[22] Meyer, B. Reusability: The case for object-oriented design. *IEEE software*, **2** (vol. 4), 1987, pp. 50.

[23] Jehan S., I. Pill, F. Wotawa. Functional SOA testing based on constraints. In *Proc of International Workshop on Automation of Software Test (AST 2013)*, 2013, pp. 33-39. doi:10.1109/IWAST.2013.6595788.

[24] Ten Dyke R.P., J.C. Kunz. Object-oriented programming. *IBM Systems Journal*, **3** (vol. 28), 2010, pp. 465-478. doi:10.1147/sj.283.0465.

[25] Paik, I., W. Chen, M.N. Huhns. A scalable architecture for automatic service composition. *IEEE Transactions on Services Computing*. **1** (vol. 7), 2012, pp. 82-95.

[26] Motogna S, A. Vescan, C. Serban, P. Tirban. An approach to assess maintainability change. *Proc. of the IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR 2016)*, 2016. doi:10.1109/AQTR.2016.7501279.

[27] Heitlager I, T. Kuipers, J. Visser. A Practical Model for Measuring Maintainability. *Proc. of the 6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*, 2007. doi:10.1109/QUATIC.2007.8.

[28] Cito, J., P. Leitner, M. Rinard, H.C. Gall. Interactive production performance feedback in the IDE. In *Proc. of IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 971-981.

[29] Winter, J., M. Aniche, J. Cito, A.V. Deursen. Monitoring-aware IDEs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 420-431.

[30] Dawes, J. H., M. Han, O. Javed, G. Reger, G. Franzoni, A. Pfeiffer. Analysing the Performance of Python-Based Web Services with the VyPR Framework. In *Proc. of International Conference on Runtime Verification*, 2020, pp. 67-86.

[31] Hasselbring, W., A. van Hoorn. Kieker: A monitoring framework for software engineering research. *Software Impact*s, 2020.

[32] Hevner A, S. Chatterjee. *Design Science Research in Information Systems*. Springer, 2010. doi:10.1007/978-1-4419-5653-8.

[33] Peffers K, T. Tuunanen, M.A. Rothenberger, S. Chatterjee. A Design Science Research Methodology for Information Systems Research. *Journal of Management of Information Systems*. **3** (vol. 24), 2007, pp. 45-77. doi:10.2753/MIS0742-1222240302.

[34] Savchenko, D., T. Hynninen, O. Taipale. Code quality measurement: Case study, *Proc. of 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, Opatija, 2018, pp. 1455-1459. doi: 10.23919/MIPRO.2018.8400262.

[35] Tufano, M., F. Palomba, G. Bavota, R. Oliveto. When and Why Your Code Starts to Smell Bad - Aditional Analysis. In *Proc. of 37th IEEE/ACM International Conference on Software Engineering*. 2015, pp. 99.

[36] Rhoton J, R. Haukioja. *Cloud Computing Architected*. Recursive Press, 2011.

[37] Robson C. *Real world research*. Blackwell Publishing, 2002. doi:10.1016/j.jclinepi.2010.08.001.

Information about the authors:

Dr. Dmitrii Savchenko – was a junior researcher with LUT Software Engineering department. Dr. Savchenko defended his doctoral thesis on the automated assessment of maintenance needs from LUT University in 2019, and currently works in the software industry as an expert and a senior developer.

Timo Hynninen – is a senior lecturer with the South-Eastern Finland University of Applied Sciences. Timo Hynninen has earlier graduated as Master of Science in technology from LUT University, and is currently finalizing his dissertation work on the automated measurement of software quality characteristics.

Professor Kari Smolander – is the current head of software engineering department at the LUT University. His research interests include, but are not limited to software processes, software architectures, enterprise architecture and software platforms.

Dr. Ossi Taipale – is an adjunct professor with LUT University. Ossi Taipale was the principal investigator of the research project .Maintain, and an associate professor with the department of software engineering. Ossi Taipale's research interest include software testing, software construction and software processes.

Assoc. prof. Jussi Kasurinen – works currently at the LUT University in the department of software engineering. Assoc. Prof. Kasurinen is also the current head of software engineering degree programs, and His research interests include but are not limited to software testing, quality assurance and software maintenance.

Manuscript received on 28 September 2020