



Timo Hynninen

DEVELOPMENT DIRECTIONS IN SOFTWARE TESTING AND QUALITY ASSURANCE



Timo Hynninen

DEVELOPMENT DIRECTIONS IN SOFTWARE TESTING AND QUALITY ASSURANCE

Dissertation for the Degree of Doctor of Science (Technology) to be presented with due permission for public examination and criticism in the Auditorium 1318 at Lappeenranta-Lahti University of Technology LUT, Lappeenranta, Finland, on the 17th of February, 2023, at noon.

Acta Universitatis
Lappeenrantaensis 1070

Supervisors Associate Professor (tenure track) Jussi Kasurinen
LUT School of Engineering Science
Lappeenranta-Lahti University of Technology LUT
Finland

Associate Professor (tenure track) Antti Knutas
LUT School of Engineering Science
Lappeenranta-Lahti University of Technology LUT
Finland

Reviewers Professor Markku Tukiainen
University of Eastern Finland
Finland

Assistant Professor (tenure track) Outi Sievi-Korte
Tampere University
Finland

Opponent Associate Professor (tenured) Daniel Russo
Aalborg University
Denmark

ISBN 978-952-335-922-2
ISBN 978-952-335-923-9 (PDF)
ISSN 1456-4491 (Print)
ISSN 2814-5518 (Online)

Lappeenranta-Lahti University of Technology LUT
LUT University Press 2023

Abstract

Timo Hynninen

Development Directions in Software Testing and Quality Assurance

Lappeenranta 2023

68 pages

Acta Universitatis Lappeenrantaensis 1070

Diss. Lappeenranta-Lahti University of Technology LUT

ISBN 978-952-335-922-2, ISBN 978-952-335-923-9 (PDF), ISSN 1456-4491 (Print),

ISSN 2814-5518 (Online)

In software engineering, testing and quality assurance activities are characterised as important yet costly phases of a product's life cycle. On the one hand, quality issues or malfunctioning products can cause expensive and potentially irreversible damage; on the other hand, rigorous quality assurance work is time-consuming and limited by the available resources. For this reason, companies aim to automate their testing and quality assurance processes as much as possible. In the modern software production environment, the use of automation, tools and even artificial intelligence is constantly evolving. Given the rapid pace of evolution, studying industry practices and observing practitioners in action is paramount for software engineering research.

This thesis investigates current practices and future development directions in testing and quality assurance work. First, a survey method is used to map the current practices. Then, the thesis utilises an empirical approach to explore novel approaches for automating quality assurance tasks. These approaches are then evaluated using the design science research method. Finally, the survey results are used to create a testing education curriculum aligned with industry practices.

As a result, the thesis presents a holistic overview of testing and quality assurance practices, tools and education. An overview of the current tools in the industry is presented, in addition to conclusions about the trends and issues related to testing. Following the issues identified in the survey, a novel tool—.Maintain—is constructed and evaluated as one solution to the runtime monitoring of software projects. The last contribution is a curriculum, learning activities and learning objectives for testing education to produce more industry-ready graduates.

Keywords: software testing, quality assurance, maintenance, survey, design science, testing education and training

Acknowledgements

First, I wish to express my gratitude toward Prof. Kasurinen and Prof. Knutas for supervising this work. It was a privilege to work with you for all these years, and I have learned a lot. I am grateful for the support and guidance you have given me throughout this journey.

Thank you to the preliminary reviewers, Prof. Tukiainen and Prof. Sievi-Korte. I am grateful for all the comments, remarks, observations, and requests to correct, which have improved the work. Thank you Prof. Russo for agreeing to be my opponent in the public examination.

Thank you, Victoria, for pushing me into finalizing this thesis. You kept me motivated and had a huge impact on the final result.

Throughout my academic career, I have met many friends and colleagues, who have had an impact on my way of thinking and ultimately the final form of this thesis. There are too many names to list but thanks go to every single person I have shared a thought with in the corridors of the university, or who has enjoyed a cup of coffee with me in the break room.

A special thank you to a few very special friends; Thank you Janne Parkkila for all the amazing work we did back in the day. Encounters with you probably set me off to seek an academic career. Thank you Lassi Riihelä and Dmitrii Savchenko for sharing a bright-minded office with me. Your knowledge and curiosity created an innovative place to work.

Finally, a big thank you to all the family who have always been so supportive. I'm grateful and proud to be a son, brother, uncle, and a godfather. The support has been invaluable.

Timo Hynninen
January 2023
Helsinki, Finland

Contents

Abstract

Acknowledgements

Contents

List of publications **9**

Nomenclature **11**

1 Introduction **13**

- 1.1 Motivation 13
- 1.2 Research approach..... 14
- 1.3 Outline of the thesis..... 15

2 Related work **17**

- 2.1 What are software testing and quality assurance?..... 17
- 2.2 Trends in software quality assurance, testing, and fault prediction 18
- 2.3 Methods for measuring software quality 19
- 2.4 Tools for analysing software quality 20
- 2.5 Software testing and quality standards..... 21
- 2.6 ISO/IEC 25010 and ISO/IEC 29119 in detail 21
- 2.7 Software testing education 25

3 Research approach and methods **27**

- 3.1 Objectives and research questions..... 27
- 3.2 Research methods 29
- 3.3 Research design..... 30

4 Overview of publications **33**

- 4.1 Publication I – Survey of the industry practices..... 33
- 4.2 Publication II – Framework for observing maintenance needs, runtime metrics and overall quality-in-use..... 37
- 4.3 Publication III – Code quality measurement: case study 41
- 4.4 Publication IV – Early-warning system for software quality issues using maintenance metrics 43
- 4.5 Publication V – Guidelines for software testing education objectives from industry practices with a constructive alignment approach..... 45
- 4.6 Publication VI – Designing early testing course curricula with activities matching the V-model phases..... 48
- 4.7 Summary of contributions 50

5 Discussion **53**

- 5.1 Research objectives 53

5.2 Findings	53
5.3 Implications for practice and research.....	55
5.4 Assessment of the research.....	56
6 Conclusion	59
References	61
Publications	

List of publications

This dissertation is based on the below original papers. The rights have been granted by publishers to include the papers in the dissertation. These publications are referred to as *Publication I*, *Publication II*, *Publication III*, *Publication IV*, *Publication V* and *Publication VI*. The author's contributions to each publication are also presented here.

- I. Hynninen, T., Kasurinen, J., Knutas, A., & Taipale, O. (2018). Software testing: Survey of the industry practices. In *Proceedings of the 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)* (pp. 1449–1454). IEEE.

Hynninen was the principal author and investigator of the paper. He carried out the surveys, analysed the data and wrote most of the article.

- II. Hynninen, T., Kasurinen, J., & Taipale, O. (2018). Framework for observing the maintenance needs, runtime metrics and the overall quality-in-use. *Journal of Software Engineering and Applications*, 11(4), 139–152.

Hynninen was the principal author and investigator in the paper. He designed the framework presented and conducted the experiments to validate the results.

- III. Savchenko, D., Hynninen, T., & Taipale, O. (2018). Code quality measurement: Case study. In *Proceedings of the 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, (pp. 1455–1459). IEEE.

Dr Savchenko was the corresponding author. Savchenko and Hynninen designed the experiments and cowrote the article. Hynninen also assembled the literature review and theoretical framework sections of the paper.

- IV. Savchenko, D., Hynninen, T., Taipale, O., Smolander, K., & Kasurinen, J. (2020). Early-warning system for software quality issues using maintenance metrics. *International Journal on Information Technologies and Security*, 12(4), 35–46.

In this paper, Dr Savchenko and Timo Hynninen designed the study and conducted the experiments. Most of the article was written by Dr Savchenko and Prof. Kasurinen. Hynninen participated in the writing and contributed to the assembly of the literature data presented in the paper.

- V. Hynninen, T., Kasurinen, J., Knutas, A., & Taipale, O. (2018). Guidelines for software testing education objectives from industry practices with a constructive

alignment approach. In *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education* (pp. 278–283). ACM.

Hynninen was the principal author and investigator of the paper. He performed the data collection and analysis and wrote most of the paper. The theoretical framework was designed in cooperation with Prof. Knutas, who also wrote the literature review section of the paper.

- VI. Hynninen, T., Knutas, A., & Kasurinen, J. (2019). Designing early testing course curricula with activities matching the V-model phases. In *Proceedings of the 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)* (pp. 1593–1598). IEEE.

Hynninen was the principal author and investigator of the paper. He performed the data collection and analysis and wrote most of the paper. The theoretical framework was designed in cooperation with Prof. Knutas and Prof. Kasurinen, who took part in writing the paper.

Nomenclature

Abbreviations

QA	Quality assurance
DSR	Design science research
SOA	Service-oriented architecture
DevOps	Development and operations
ISO	International Organisation for Standardisation
MI	Maintainability index
IDE	Integrated development environment

1 Introduction

Across the software business sector, software testing is an important part of quality assurance (QA). QA activities in general aim to ensure that the products and services offered are of the best possible quality. Testing efforts aim to ensure that products are shipped with no or few defects.

Testing is arguably a difficult part of the software development process. Whittaker (2000) describes testing as the least understood part of development. Often, testing and QA work are automated as much as possible, and for this reason, there exists a plethora of different testing tools, technologies and frameworks for test automation (Prasad et al., 2021). However, the complex nature of testing work, in addition to the number of different tools used in the industry, can cause configuration problems (Kasurinen et al., 2010).

Good testing practices and QA processes can reduce the total costs of the software life cycle by reducing the number of defects during development. However, in the overall life cycle models for software, this is only the first step in a lengthy phase: maintenance. The maintenance phase of software is generally considered the biggest overall expense. Therefore, QA and testing activities must also be carried out after any dedicated development or testing phase.

The tools and processes used in the industry constantly evolve, so much so that industry practices and academic research are ‘worlds apart’ (Garousi & Felderer, 2017). This evolution has brought more sophisticated tools and automation to reduce the costs related to testing and maintenance. Working in this context, the objective of the current thesis is to explore the different development directions related to software testing and QA work.

The current study investigates testing practices in the industry and proposes solutions to some of the problems plaguing the business sector. The first contribution of the study to the state-of-the-art is a survey mapping the current testing and quality assurance practices in the industry. Two other contributions are also presented: Tools for measuring software quality characteristics, and recommendations for learning objectives in testing education.

1.1 Motivation

By definition, software testing is the activity conducted to establish and assess the quality of software products (Osterweil, 1996). Myers et al. (2004) offer a more pragmatic view with the definition of testing as ‘the process of executing a programme with the intent of finding errors’. In practice, software testing activities cover most of the QA work (Kasurinen, 2013).

Testing is characterised as an activity simultaneously expensive and money-saving. On the one hand, testing is a costly activity (Garousi, Arkan, et al., 2020), and it is often not conducted efficiently (Taipale & Smolander, 2006). In 2013, the price of finding and fixing software defects was estimated to be US\$312 billion globally (Britton et al., 2013).

The high price tag of testing is largely because of the high cost of fixing defects after the design and development phases have been completed (Kit, 1995; Planning, 2002). The costs related to poor quality products, such as malfunctioning programmes and errors in functionality, cause large expenses.

On the other hand, QA work can save money in the long run. At the beginning of the millennium, a US National Institute of Standards and Technology report estimated that US\$21.2 billion of direct losses could have been prevented (Tasse, 2002). The same report estimated that an additional US\$59.5 billion could be saved when accounting for indirect losses to clients and customers. More recently, studies have indicated that the costs related to testing are on the rise. The software industry has identified a need to reduce the growing cost of test management (Capgemini, 2017).

In addition, the rise of software-as-a-service distribution methods (Ma, 2007) and continuous delivery models (Chen, 2015) has made the maintenance phase one of the most costly in the life cycle of a software product (Capgemini, 2017; Kyte, 2012). In some software industries, the first launch expects the system to include only the bare essentials, and the majority of the content is developed while the system itself is in ‘the maintenance phase’ (Leppänen et al., 2015). However, few software development models or software process models consider changing deployment practices.

Testing practices and processes in software are usually ad-hoc (Garousi, Arkan, et al., 2020). In addition, testing is often manual work that relies on the experience (and creativity) of the testers (Myers et al., 2004). Although many software companies have established processes for testing and quality control, many studies have suggested there is room for improvement in industry practices (e.g., Garousi et al., 2015; Garousi & Zhi, 2013; Garousi & Varma, 2010; Taipale & Smolander, 2006). Given how expensive testing and QA work is, improvements in this line of work could bring significant savings while improving product quality. A recent study by Wang et al. concludes that ‘there is lack of guidelines on designing and executing automated tests and the right metrics to measure and improve test automation processes in general’ (2020).

As the costs of testing are on the rise (Capgemini, 2017), and the software industry could benefit from research and development efforts (Garousi, Arkan, et al., 2020), there is a need for further empirical study on the practices, processes, and tools related to testing. Academia can benefit from a better understanding of industry practices. Aligning research with industry practices can also be used to improve education and training, and produce more knowledgeable and industry-ready graduates.

1.2 Research approach

The current thesis investigates software testing through the lens of monitoring software during the maintenance phase. The research presented falls under the umbrella of software testing and QA. Specifically, the research examines the practices in the industry, reveals

development directions in the area, presents new tools for practitioners and examines the education and competencies related to testing and QA.

The thesis utilises both quantitative and qualitative research methods. First, the survey method is used to study testing and QA practices in the Finnish industry. Current industry practices in testing, processes, tools and automation are investigated to obtain an overview of the state of the art. Next, the current study focuses on tools for monitoring software quality and detecting defects. This, in turn, is achieved by applying the design science research method. Finally, testing education and training are explored in the same context. Constructive alignment is used as the main method in the final phase of the research.

1.3 Outline of the thesis

The present thesis is divided into two parts: an introduction and six scientific publications as an appendix. The introduction outlines the general research area, the research approach, including research questions and research process, and synthesises the overall results from the scientific publications. The appendix contains six publications, which describe in detail the individual research studies that form the research programme outlined in the thesis.

2 Related work

This section presents the literature and central concepts related to software quality. First, a literature review is presented to provide an overview of the field. Next, the methods and tools for quality assurance (QA) are discussed. Then, related software testing and quality standards are presented. Finally, extant literature related to testing education is discussed.

2.1 What are software testing and quality assurance?

Software testing provides information about the quality of the software (Kaner, 2006). Testing consists of verification and validation work (Kit, 1995). Verification means the evaluation of the product's compliance with certain requirements (IEEE, 2011). Validation, on the other hand, means the assurance that the product meets the needs of the customer and other stakeholders (IEEE, 2011). In layman's terms, verification answers the question 'Are we building the product right?' whereas validation answers the question 'Are we building the right product?'

Testing is often defined as the process of finding faults in a software product. The most traditional—and arguably the most pragmatic—definition is by Myers et al. (2004): 'Testing is the process of executing a programme with the intent of finding errors'. This definition reflects the verification aspect of testing. A broader definition for testing is offered by the joint ISO/IEC and IEEE standards as 'activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component' (ISO/IEC, 2017).

Many definitions tend to share Myers' view but augment it by addressing validation as well; for example, Whittaker's (2000) definition of testing as 'the process of executing a software system to determine whether it matches its specification' is almost identical, except that it focuses on validation instead of verification. In addition to focusing only on either verification or validation, the previous definitions by Myers et al. and Whittaker are quite narrow also because they do not cover static features of software—for example, code reviews would not be considered testing under these definitions.

The ISO/IEC 29119 standard for software testing defines testing from the viewpoint of the testing process. That is, testing is comprised of the individual processes and overarching organisational policies. The standard covers, for example, dynamic test processes, static test processes, test management, test monitoring and control, test strategy and test policy (ISO/IEC, 2013).

Quality assurance is related to all the processes in software development that aim to improve product quality. According to ISO 9000, QA is the 'part of quality management focused on providing confidence that quality requirements will be fulfilled' (ISO/IEC, 2005). Software testing covers most of QA (ISO/IEC, 2013), so the two concepts have a strong connection.

2.2 Trends in software quality assurance, testing, and fault prediction

Many recent studies have focused on the trends of QA, testing, and fault prediction, including meta-analyses. For example, Catal (2011) performed a literature survey containing 90 academic papers about software fault prediction published between 1990 and 2009. The survey covered statistics-based and machine-learning-based approaches for fault prediction. The publication trends indicate that fault prediction gained interest during the two decades of observation. The survey also highlights significant researchers in the field and practical software fault prediction problems identified in the literature. The results highlight the need for software assessment models because machine learning and data analytics approaches to fault prediction require an extensive amount of available data. Another challenge identified regarding the lack of data is the validity of the constructed assessment models when there is not enough fault data to build accurate models.

Goues et al. (2013) identify several factors in programme design that are critical to maintenance. The article provides a high-level overview of the state of the art in automatic programme repair. The identified programme design issues affecting scalability and repair success include focusing on the most visited (high-risk) code areas and feature development in patches (adding functionality in the late stages of the development cycle). The challenges in automated programme repair are identified as locating possible fixes, evaluating repair quality, the absence of complete test suites or formal specifications and accepting the change or new ways to work.

Sarwar et al. (2008) analyse and compare different tools for calculating the maintainability index (MI). The article highlights that each tool has its strengths and weaknesses, hence producing a different MI score in different circumstances. As a result, the article calls for standardisation of MI calculation formulas and more open-source tools to support maintainability evaluation.

These studies suggest, that there is a need for further study in fault prediction and early-warning systems for quality issues. The detection of faults and high-maintenance software modules is an important research avenue. Existing measures can be utilised but more high-level frameworks could be developed.

2.3 Methods for measuring software quality

Many studies have approached software quality through maintainability characteristics. One of the earliest in this line of work is the study by Lewis and Henry (1989), in which the authors present a method for integrating maintainability into large-scale software projects. Many recent studies follow the ideas presented by Lewis and Henry, the most notable of these being the concept of using code metrics as an indicator of quality.

In the study by Koru and Tian (2005), high-change code modules are identified. Two large-scale open-source products, Mozilla and OpenOffice, are used. The study compares the high-change modules with the modules with the highest measurement values. The authors conclude that, although high-change modules also have high measurement values, they are, however, not the highest scorers in the code quality metrics.

Ghods and Nelson (1998) evaluate the factors that contribute to quality during the maintenance phase of the software life cycle. The study emphasises that design choices towards better maintainability positively impact quality during the maintenance phase. The results indicate that quality during maintenance results from good application design combined with a strong bond between software maintainers and end users.

In a similar vein, Yamashita (2015) performs software quality evaluations by combining metrics analysis, software visualisation and expert assessment techniques. The research presents a case study of a software quality evaluation process performed for a logistics company. The results show that automatic software benchmarking provides useful information to aid in decision making, but at the same time, it should be complemented with inspections and visual analysis.

Hegedus (2013) studies the effect of coding practices on maintainability. This work examines two Java-based systems using a probabilistic measurement model. The results indicate a strong correlation between the density of design patterns in code and maintainability of a system. The software measurement model combines static code complexity metrics, for example, McCabe metrics, with in-use metrics measuring fault proneness.

Janus et al. (2012) introduce continuous measurement and continuous improvement into the development process as subsequent activities to continuous integration. This article establishes software quality metrics for an agile development process. The approach is then validated in a legacy web application project.

Herzig et al. (2015) present a generic test selection strategy that aims to improve the agility of development. The test selection method is based on the cost estimation of

running a test, and it removes tests from a suite when the expected cost of running a test exceeds the cost of removing it. The article describes a cost model for test executions, which is then evaluated using large projects, such as Microsoft Office or Windows.

Rompaey et al. (2007) introduce a conceptual model for software testing. Based on the model, the article proposes metrics that are used to measure smells in unit tests. As a result, the article demonstrates how the proposed metrics can be used to automate test evaluation.

Although these studies investigate how to measure the quality of software code, they focus on individual characteristics such as performance, maintainability, security, or usability. Thus, there is a need for further investigation toward a framework that combines multiple quality characteristics. Such a framework can use the proven methods of measuring software quality and further extend their utility.

2.4 Tools for analysing software quality

Some of the previous studies in the field of measuring software quality also include tools that can be used to adopt the methods. For example, Motogna et al. (2016) present an approach to measure software maintainability by using the characteristics of maintainability as defined by ISO/IEC 25010. Their analysis maps object-oriented metrics to maintainability characteristics from the software quality model. This work shows the influence of code metrics on quality characteristics and how different metrics affect maintainability subcharacteristics.

CODEMINE is a data analytics platform for collecting and analysing data related to engineering processes at Microsoft. The platform collects metrics from source code repositories, reports, test and deployment platforms and project management systems. It can be used for onboarding processes, optimising individual processes and optimising code flow (Czerwonka et al., 2013).

PerformanceHat is a plug-in for the Eclipse integrated development environment (IDE). The objective of PerformanceHat is to analyse performance problems in software projects by integrating analytics directly into the IDE. The studies by Cito et al. (2018, 2019) on the tool show that developers who use it are faster at detecting problems and better at finding the cause of those problems.

Suliman et al. (2006) present a built-in test infrastructure, where component testing is realised using a test responsibility approach. The article describes an infrastructure that supports testing at runtime through features such as test isolation, test scheduling and resource monitoring.

2.5 Software testing and quality standards

In the current study, the ISO/IEC 25010 (ISO/IEC, 2011b) software quality and ISO/IEC 29119 (ISO/IEC, 2013) software testing standards serve as the theoretical framework for the technological aspects. The present work is mainly based on the ISO 25010 System and software quality models because they are relatively recent yet well-established standards. ISO 25010 provides a comprehensive model for a quality measurement framework in its quality-in-use model. Additionally, the standard provides examples of how to derive metrics and execute measurements.

Standards can provide a rough overview of technological fields, such as testing and QA. However, often standards are too generic for practical use, as they are generally quite high-level. As such, standards are a good starting point for an evaluation of software quality, maintainability or complexity, but solutions that implement the ideas in these frameworks are scarce.

In the literature, many software measurement frameworks are based on—or at least influenced by—the ISO/IEC quality models. Examples of these studies include the software maintainability measurements developed by Motogna et al. (2016), the longitudinal project to evaluate the maintainability of software projects by Molnar and Motogna (2020), the performance measurement framework for cloud computing by Bautista et al. (2012) or the framework for evaluating the effect of coding practices on software maintainability by Hegedus (2013). Thus, the ISO 25010 standard is often used in software engineering research, which makes it a good starting point for the current study.

However, previous research has been limited to covering only parts of quality models. Previous studies have concentrated on specific quality characteristics such as maintainability or performance efficiency. To the best of the author's knowledge, no research or tool exists where the entire software quality model has been considered. There is a need for further work with a general measurement framework and tools, in which the aim is to incorporate the characteristics of a software quality model into a software measurement tool.

2.6 ISO/IEC 25010 and ISO/IEC 29119 in detail

ISO/IEC 25010 describes the quality for software as a combination of 'system/software quality' and 'system/software quality-in-use'. Software quality consists of those characteristics related to the design and implementation of software, whereas quality-in-use is described by the characteristics related to the outcomes of the interaction with the software. The ISO/IEC 25000 standard aims to clarify the requirements for assessing software quality. Thus, the ISO/IEC 25010 quality model aims to depict the software system as a complete computer-human system, in which systems have both static properties and interaction with users (ISO/IEC, 2011b).

The software and system product quality characteristics are divided into eight categories, which are further divided into 31 subcharacteristics. The main characteristics are functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability and portability. The characteristics focus on the technical aspects of the software, even though there are also more human-centric aspects, such as learnability or aesthetics. The ISO/IEC 25000 standard family also contains recommendations for measuring the characteristics and subcharacteristics using quantitative and qualitative metrics. Figure 2.1 shows the software product quality attributes.

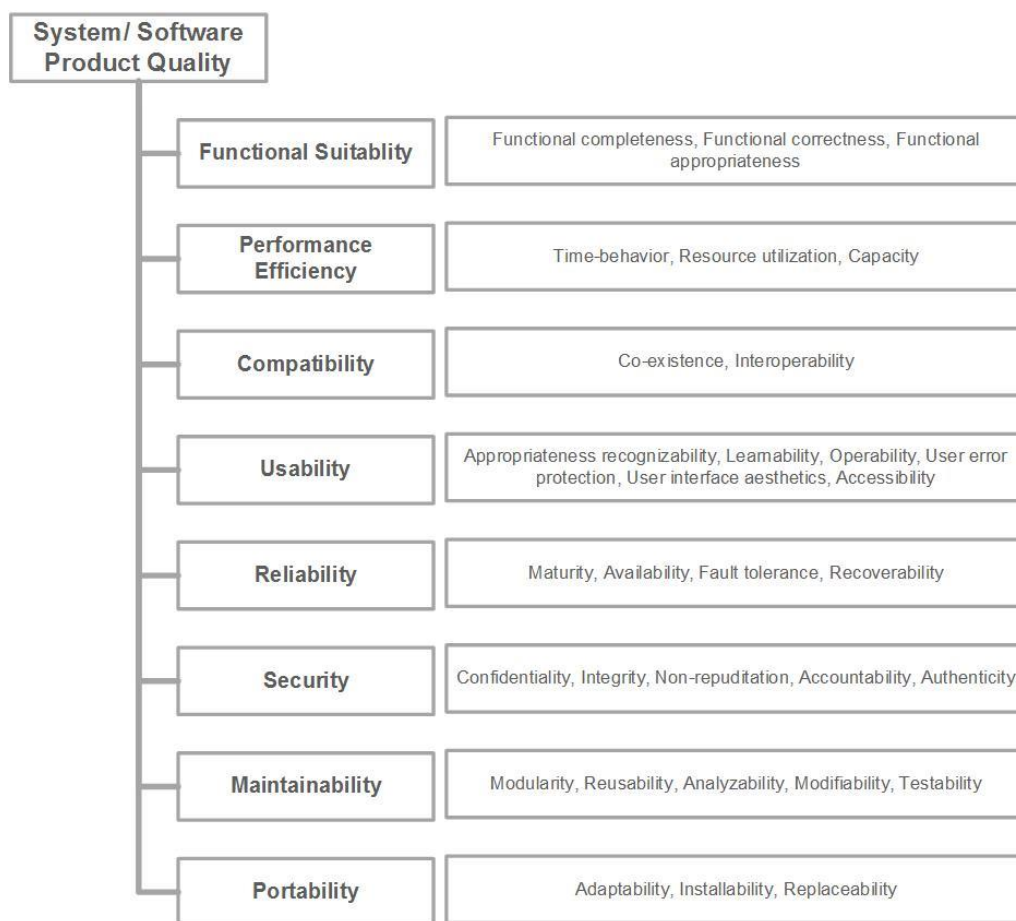


Figure 2.1: The ISO/IEC 25010 software / system product quality model (adapted from (ISO/IEC, 2011b)).

The quality-in-use model includes five main characteristics, which are further divided into 11 subcharacteristics. The quality-in-use model focuses on the effect of the use of the software and user experience. Unlike the product quality model, several of the subcharacteristics are recommended to be measured using psychometric scales or other user-centric methods. Figure 2.2 presents the quality-in-use attributes in detail.



Figure 2.2: The ISO/IEC 25010 software quality-in-use model (adapted from (ISO/IEC, 2011b)).

In the current thesis, the examination of software quality is related to the maintenance phase of the software life cycle. On its own, maintainability is one of the eight product quality properties defined in ISO/IEC 25010. Maintainability is the ‘*degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers*’ (ISO/IEC, 2011b). Maintainability has also previously been defined as ‘*the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment*’ in software standards concerning software life cycle processes (ISO/IEC, 2006).

Maintainability in the ISO 25010 software quality model consists of **modularity** (‘*degree to which a system or computer programme is composed of discrete components*’), **reusability** (‘*degree to which an asset can be used in more than one system*’), **analysability** (‘*degree with which it is possible to assess the impact on a product or*

system’), **modifiability** (‘degree to which a product or system can be modified’) and **testability** (‘degree of effectiveness and efficiency with which test criteria can be established for a system’) (ISO/IEC, 2011b).

Table 2.1: Verification and validation activities (adapted from (ISO/IEC, 2013))
Verification and validation activities as categorised in ISO/IEC 29119

Testing	Static testing	<i>Inspection</i>
		<i>Reviews</i>
		<i>Model verification</i>
		<i>Static analysis</i>
	Dynamic testing	<i>Specification-based methods</i>
		<i>Structure-based methods</i>
		<i>Experience-based methods</i>
Formal methods	Model checking	
	Proof of correctness	
Verification & Validation Analysis	Simulation	
	Evaluation	Quality metrics

As far as testing and QA activities are concerned, the ISO/IEC 29119 standard focuses on testing and the test process itself (ISO/IEC, 2013). This standard aims to form a consensus regarding what areas software testing includes and to serve as a reference manual for common concepts and definitions. Additionally, the standard can be employed as a reference model for software testing activities at the guidance level.

In this standard, testing is seen in the context of verification and validation. This is because most verification and validation activities are covered by testing. As a whole, testing is presented as a risk-based activity because this approach allows testing to be prioritised and focused. The classification of the different verification and validation activities, as depicted in the ISO/IEC 29119, which is presented in Table 2.1.

In particular, the standard covers dynamic and static testing methods. Static testing refers to the examination of programme code through *inspections*, *code reviews*, *model*

verification and *static analysis* methods. The objective of static testing is to ensure that the programme code does not contain faults or errors, and that the code is written with readability in mind.

In turn, dynamic testing refers to the techniques used to examine the programme's behaviour. Specification-based methods, for example *equivalence partitioning* or *boundary value analysis*, use specifications (such as requirements or models) as the basis for the test conditions. In structure-based methods, such as *branch testing*, the structure of the programme (commonly source code but also models of the system) is used to design test cases.

Experience-based methods, such as *error guessing*, differ from specification- and structure-based methods in that they rely on the experience of the tester or test designer to identify fault-prone components and common errors. The different testing methods are complementary, and usually, a combination of all is required for effective testing.

2.7 Software testing education

The literature related to testing education has previously been synthesised in the literature reviews by Desai et al. (2008), Scatalon et al. (2019), Lauvås Jr & Arcuri (2018), and Garousi et al. (2020).

Lauvås Jr & Arcuri (2018) conducted a literature review on the recent trends in testing education. According to the results, most studies focus on the pedagogical approaches to teaching testing. Additionally, many studies present software and tools to support teaching. Most of the existing work in testing education describes experiences, and not many meta-analyses have been conducted on the topic.

Garousi et al. (2020) performed a literature mapping study. According to the results, the main activities that are present in testing courses are generic software testing, test-case design, test automation, and test execution. In addition, the study points out some of the challenges in teaching software testing, including motivating students, time and resource requirements for the instructors, the complexity of the topic, and alignment with the industry needs.

In terms of how the testing skills differ between students (novices), and professionals, the study by Bai et al. (2021) found that students tend to struggle with test coverage and generally have poor knowledge of how to write good tests. Writing tests also help students write better code (Lazzarini Lemos et al., 2017; Lazzarini Lemos et al., 2015; Scatalon et al., 2017). Instructors can also employ checklists to guide the students through designing tests (Bai et al., 2022).

Extant literature shows a consensus that testing education produces more knowledgeable and industry-ready graduates. At the same time, there are also challenges related to testing as the teaching topic. For example, students' motivation to study testing is a recurring

theme (Garousi et al., 2020; Lauvås Jr & Arcuri, 2018). This suggests, that there is a need for further studies on the design and validation of testing curricula. One viable approach is to integrate real-world contexts in the testing courses (Krutz et al., 2014; Lopez et al., 2015; Valle et al., 2017). Thus, there is room for the development of testing curricula that are aligned with industry practices.

3 Research approach and methods

This chapter describes the research approach and methods used in the present dissertation. First, the objectives of the work and overarching research questions are presented. Next, the research method and design are detailed. Finally, the different stages of the work are discussed. The research process was split into four phases. Table 3.1 provides an overview of the phases, research approach, outcomes and relation to the included publications.

3.1 Objectives and research questions

The current thesis aims to investigate development directions in testing and QA. The current study focuses on the testing practices, processes and tools of the industry, existing quality measurement standards, as well as the education of software engineers. The main research question is as follows: **To what extent can test automation and software measurement tools improve testing and QA work in software companies?** The main research question is further divided into the following subquestions:

RQ 1: What is the current state of industry practices in testing and QA, and how have they evolved in recent times?

RQ 2: What kind of framework would enable measurement of software quality characteristics and detecting maintenance issues?

RQ 3: To what extent can runtime quality metrics be collected from real software projects to analyse quality and maintainability?

RQ 4: To what extent are software engineers ready to use the testing and QA tools, and how can testing education be better oriented to support this goal?

Table 3.1: Overview of the research approach

	Phase 1	Phase 2	Phase 3	Phase 4
Phase objective	Understand the current state of industry practices in testing and QA.	Investigate how QA activities could be improved in the maintenance phase of the software life cycle.	Design, implement and validate tools that automate the collection of software quality metrics.	Evaluate the competencies and education of software engineers.
Research questions	What is the current state of the industry practices in testing and QA, and how have they evolved in recent times? (RQ 1)	What kind of framework would enable measurement of software quality characteristics and detecting maintenance issues? (RQ 2)	To what extent can runtime quality metrics be collected from real software projects to analyse quality and maintainability? (RQ 3)	To what extent are software engineers ready to use the testing and QA tools, and how can testing education be better oriented to support this goal? (RQ 4)
Method	Survey	Design science	Design science	Survey; Constructive alignment
Outcomes	Survey mapping the industry practices in testing and quality assurance	Design and implementation of a framework for runtime software measurement	Design, implementation, and evaluation of a tool for measuring software quality characteristics	Curriculum and learning objectives for testing education aligned with industry practices
Related publications	Publication I	Publication II	Publications III and IV	Publications V and VI

3.2 Research methods

This section covers the selection of the research methods used in the current thesis. The selected research approach combines quantitative and qualitative research methods. First, the survey method, as a quantitative research approach, is used to provide an overview of the field. Next, qualitative approaches are employed to design, build and evaluate artefacts.

Survey method

The survey method was used at the beginning of the research programme. Fink and Kosecoff (1985) describe the objective of a survey as collecting information from people about their feelings and beliefs. Surveys are the most appropriate when information comes directly from people (Fink & Kosecoff, 1985).

Surveys can be employed as a data collection method for both descriptive and prescriptive studies. Descriptive studies aim to produce descriptive theories (or kernel theories) based on existing theories and new data (Fischer et al., 2010). Prescriptive studies, including design science research, use data to construct useful artefacts (Carstensen & Bernhard, 2019), including models, methods, constructs, instantiations and design theories (March & Smith, 1995; March & Storey, 2008).

Multiple approaches exist for survey research design. In the present work, a cross-sectional research approach to the survey method was employed. In cross-sectional survey studies, a relevant sample of a population is drawn and studied (Shaughnessy et al., 2012). Cross-sectional studies provide descriptive statistics of the target population at one time, but they cannot be used to draw conclusions about the factors explaining the results (causation).

The survey research conducted within the current thesis has been positioned as exploratory, observational and cross-sectional work exploring practices in the software industry.

Design science research

The design science research (DSR) method (Gregor & Hevner, 2013; Hevner et al., 2004; Hevner, 2007; Peffers et al., 2007) is an outcomes-based research method providing a framework for the design, implementation and evaluation of systems and artefacts. Hevner and Chatterjee define DSR as ‘a research paradigm in which a designer answers questions relevant to human problems via the creation of innovative artefacts, thereby contributing new knowledge to the body of scientific evidence. The designed artifacts are both useful and fundamental in understanding that problem’ (2010).

The iterative approaches employed in DSR can enable the development of different artefacts, ranging from theories (Kuechler & Vaishnavi, 2008) to engineering designs and models (Carstensen & Bernhard, 2019). The DSR approach was selected to support the

design and implementation of the software tools created as part of the research programme. The objective was to design and implement tools that automate the collection of software quality metrics. DSR provided a research methodology for the empirical work related to software development and a framework for the evaluation and dissemination of the results.

In DSR, the novelty of artefacts can be seen through the lenses of applicable knowledge and business needs. Rigour in the process is demonstrated through the application of existing theories and methodologies. Relevance relates to the existence and fulfilment of business needs, which can be demonstrated by applying the artefact in a real-life environment (Hevner et al., 2004).

Constructive alignment

Constructive alignment is an outcome-based approach to education. In constructive alignment, the learning outcomes that students are intended to achieve are defined in advance. Teaching and assessment methods are then designed to best achieve preset outcomes (Biggs, 1996, 2014). Hence, constructive alignment is suitable for pedagogic design, where the teaching topics follow established industry practices.

The current study employed constructive alignment as the main method for exploring the activities and learning objectives of a testing curriculum.

3.3 Research design

Finally, the research design in the current thesis is described in detail. Each phase consisted of an independent objective, research question, research methods and outcomes. Both quantitative and qualitative approaches were employed, with the whole work consisting of survey research, design science and constructive alignment. The following presents a breakdown of each phase of the research programme.

Phase 1

In Phase 1, a survey method (Fink & Kosecoff, 1985) was used to elicit information from professionals working in software development companies. The responses were detailed on the level of organisational units. This led to an analysis of how software organisations test their products and what process models they follow. Additionally, the collected data were compared with prior surveys to understand how industry practices have changed.

The survey instrument included questions about software development and QA practices, tools and challenges related to QA. The present study investigated the use of test automation, test infrastructure, agile practices and formal process models. The results of Phase 1 are documented in *Publication I*.

The results from the survey were used as the first step towards understanding contemporary testing and QA challenges in the software development process. These results helped form a picture of the types of automation currently in use in the software industry and the interest in tools that could be further explored in future research. Later, the survey was used to align testing education with industry practices.

Phase 2

In Phase 2, the design science research (DSR) method (Gregor & Hevner, 2013; Hevner, 2007; Hevner et al., 2004; Peffers et al., 2007) was the primary approach. The design science approach was selected because it is particularly suitable for engineering problems (Hevner et al., 2004; Peffers et al., 2007). DSR uses an iterative design process to create artefacts to solve a specific problem. The outcome of the design process is then rigorously evaluated in practice. The research process is considered successful if the artefact quantifiably solves the problem (Hevner et al., 2004).

This approach was used to design and implement a framework for runtime software measurement. The design and evaluation of the tools were carried out following an iterative process, and the results of the first iteration are documented in *Publication II*. The utility of the framework was demonstrated using descriptive scenarios and use cases. However, as the DSR method requires rigorous evaluation in practice, further refinement of the framework continued in Phase 3 of the current study.

Phase 3

The DSR approach was continued in Phase 3. This phase consisted of the design, implementation and evaluation of a tool for measuring software quality characteristics. The design of the tool was based on the runtime software measurement framework designed earlier. Following the principles of DSR, the construction of the tools was documented in *Publication III*, while *Publication IV* presents the evaluation and proof of utility.

In *Publication III*, the focus was on the design, construction and initial evaluation of a tool for analysing and visualising the maintainability of a software project. This work consisted of designing the software architecture for the maintenance metrics collection and analysis software, hence demonstrating the rigour of the work, as necessitated by the DSR method.

Publication IV presented the *.Maintain* (read: *dot maintain*) tool for measuring the quality characteristics of a software product. The design, architecture and operating principles of the tool were demonstrated, along with use cases and descriptive scenarios. The utility of the tool was demonstrated by presenting a case study where working software products were used as a proving ground for the tool. In the case study, the metrics provided by the tool were collected and reviewed in an in-depth interview with a project manager/product

owner. The case study provided a real-life environment through which the relevance of the tool could be demonstrated.

Phase 4

In Phase 4, the research aimed to support an understanding of software testing for new professionals through education and training. The results from the survey in Phase 1 were used to plan a contemporary testing curriculum. The primary research method was constructive alignment (Biggs, 1996, 2014).

The outcome of this phase was a curriculum and learning objectives for testing education. The curriculum design used the constructive alignment approach to fit the learning objectives, together with current industry practices. The design and evaluation of the curriculum are documented in *Publications V* and *VI*.

4 Overview of publications

This chapter presents an overview of the publications included in the thesis. The full publications are included in Appendix 1. In this chapter, the publications are summarised in terms of their research setting, methodology, results and relation to the entire thesis.

4.1 Publication I – Survey of the industry practices

Background and objectives

Testing can be one of the most expensive tasks for software projects. Besides causing immediate costs, problems with testing are also related to the costs of poor quality, malfunctioning programmes and errors, all of which can cause large additional expenses to software producers during maintenance (Kit, 1995; Planning, 2002). The costs related to testing are on the rise; the software industry has identified a need to reduce the growing cost of test environment management (Capgemini, 2017).

The objective of *Publication I* was to explore the testing practices of software companies. To achieve this, we used an online survey, in which we collected responses from people working in 33 different software companies. Additionally, the survey responses were compared with the results of a similar survey conducted nine years earlier in 2009 (Kasurinen et al., 2010), which itself was a follow-up survey to one in 2005 (Taipale et al., 2005).

Results and contributions

In this study, we surveyed organisational units (OU) representing different sizes and business domains in software development. The survey questionnaire consisted of multiple choice, multiple item questions to collect quantitative data for statistical analysis and open-ended questions for qualitative analysis.

The study mapped the utilisation of different testing tools used in the industry and current problems relating to testing and tools of the trade. The results are summarised in Tables 4.1 and 4.2, respectively.

Additionally, we compared the survey results to the results of a similar survey conducted in 2009. The comparison revealed changes in industry practices. Finally, the survey also contained a self-assessment of the quality of the different testing and QA practices, which we were also able to compare to earlier survey results. These results are presented in Table 4.3.

The results show that organisations have shifted towards automation in testing, moving away from manual testing. They have taken advantage of more sophisticated testing infrastructures, applied more agile practices even in mission-critical software and reduced the use of formal process models.

This study set the foundations of the thesis. The results enabled us to understand the current industry practices in software testing. Tools and practices in testing were further explored in subsequent publications.

Table 4.1: Percentage of the testing and QA tools utilised in the industry, as identified in our 2017 survey and previously in 2009 (Kasurinen et al., 2010).

Tool	% of respondents	
	2017	2009
Bug/defect reporting	72.7%	22.6%
Test automation	66.7%	29.0%
Unit testing	57.6%	38.7%
Bug/code tracing	57.6%	3.2%
Performance testing	48.5%	25.8%
Test case management	45.5%	48.4%
Integration testing	45.5%	16.1%
Virtual test environment	42.4%	12.9%
Quality control	36.4%	19.4%
Automated metrics collector	36.4%	3.2%
System testing	27.3%	9.7%
Security testing	24.2%	3.2%
Test completeness	24.2%	6.5%
Test design	15.2%	22.6%
Protocol/interface conformance tool	9.1%	6.5%

Table 4.2: Software test process problems, as identified in our 2017 survey and previously in 2009 (Kasurinen et al., 2010). Responses are on a scale of 1 to 5 (1 – fully disagree, 3 – neutral and 5 – fully agree).

	2017 mode	2009 mode
Complicated testing tools cause test configuration errors.	4	1
Commercial testing tools do not offer enough support for our development platforms.	3	1
It is difficult to automate testing because of its low reuse and high price.	4	5
Insufficient communication slows the bug-fixing and causes misunderstanding between testers and developers.	4	2
Feature development in the late phases of the product development shortens testing schedule.	4	4
Testing personnel do not have expertise in certain testing applications.	4	4
Existing testing environments restrict testing.	3	4

Table 4.3: The self-assessment of the quality of the different testing and QA practices, as identified in our 2017 survey and previously in 2009 (Kasurinen et al., 2010). Responses are on a scale of 1 to 5 (1 – fully disagree, 3 – neutral and 5 – fully agree).

	2017 mode	2009 mode
Our software correctly implements a specific function. We are building the product right.	4	5
Our software is built traceable to customer requirements. We are building the right product.	5	4
Our formal inspections are OK.	4	2
We go through checklists.	2	3
We keep code reviews.	1	4
Our unit testing (modules or procedures) is excellent.	4	2
Our integration testing (multiple components together) is excellent.	3	3
Our usability testing (adapt software to users' work styles) is excellent.	3	2
Our function testing (detect discrepancies between a programme's functional specification and its actual behaviour) is excellent.	3	4
Our system testing (system does not meet requirements specification) is excellent.	3	4
Our acceptance testing (users run the system in production) is excellent.	4	4
We keep our testing schedules.	2	4
Last testing phases are kept regardless of the project deadline.	4	4
We allocate enough testing time.	2	4

4.2 Publication II – Framework for observing maintenance needs, runtime metrics and overall quality-in-use

Background and objectives

Postrelease maintenance is usually the most expensive phase in the software product life cycle, which cover the first design concepts to the end of product support. Knowing this, it is rather surprising that the software development processes do not focus more on the maintenance phase. Instead, development processes focus on enhancing and offering product quality and quality-in-use improvements within the development and QA steps. For example, the Scrum software process model, which is favoured in many organizations does not take into account any activities that happen before or after active sprints, even though most software-related costs are not realised within this period.

The objective of *Publication II* was to study the different methods of monitoring software in the maintenance phase. We hypothesised that lowering the amount of work required for maintenance by predicting and identifying the changes in the quality characteristics could reduce the costs of maintenance. Thus, the aim was to build a software quality measurement framework into the source code as a library of measurement tools.

Changes in quality measures serve as an early-warning system of problematic components and software failures. More specifically, we concentrated on developing a library of software measurement probes using the ISO/IEC 25000 standard of software quality attributes as a starting point. The research questions in *Publication II* were as follows: What kind of technical infrastructure would enable identification of online quality characteristics and thereby maintenance issues? How can a software quality model be incorporated into a library of runtime metrics?

Results and contributions

In *Publication II*, our approach was to define a framework and implement the framework in a system to collect and monitor runtime data from an open-source application. In addition, the collected data are visualised with a separate analysis tool to monitor the trends and changes between the different versions of the system and assess, for example, resource usage for the customer environments.

The study presented the implementation of a framework for software measures and a proof-of-concept prototype using an open-source project. The framework can provide a systematic interface that can be used to collect runtime metrics and measure software quality-in-use. The developed software metrics are presented in Table 4.4.

The measurement framework and proof-of-concept project were evaluated using descriptive scenarios for software in the maintenance phase of its life cycle. For example, Figure 4.1 shows a time-performance metric collected from six different test scenarios. Slowness or times when an application becomes unresponsive can be detected using this

measure. Similarly, Figure 4.2 presents a utilisation metric collected demonstrating how users adopt new functionality in software. The measurement framework was implemented as a metrics library, and measurements were linked to the software during development. This work mapped runtime software metrics to quality characteristics.

In summary, the study presented a framework for runtime software measurement. The framework aimed to be general to warrant use in different applications but at the same time loose enough to allow developers to derive application-specific measurement. This contributed to the field of source code modelling and defect prediction methods. The designed framework extended the state of the art by developing concrete metrics that could be used to automate the measurement process.

Table 4.4: Ways to measure the different quality characteristics in the proof-of-concept environment.

ISO 25010 Quality Characteristic (Subcharacteristic)	Ways to measure in the framework
Functional suitability (functional correctness, functional appropriateness)	Code coverage, user-applied action to achieve use case outcomes
Performance efficiency (time behaviour)	Mean response time, response time adequacy, mean throughput
Compatibility (interoperability)	External interface adequacy
Usability (learnability)	Error messages understandability, user error recoverability
Reliability (maturity)	Mean time between failure (MTBF), failure rate
Security (accountability)	System log retention
Maintainability (analysability, modifiability)	System log completeness, modification correctness
Portability (adaptability)	Operational environment adaptability
Effectiveness	Task error intensity
Efficiency	Task time
Satisfaction	Feature utilisation
Freedom from risk (economic risk mitigation)	Business performance, errors with economic consequences
Context coverage (flexibility)	Proficiency independence

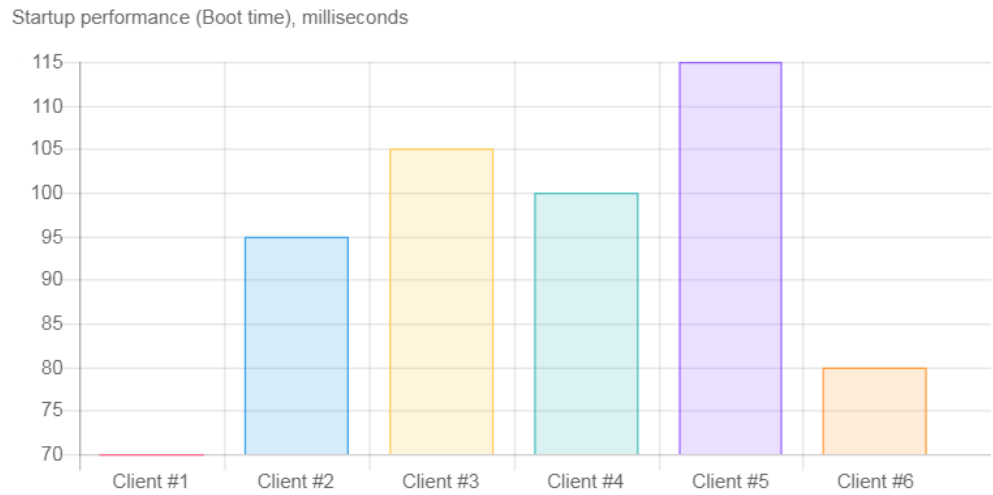


Figure 4.1: A time-performance metric collected from six different clients in a test scenario.

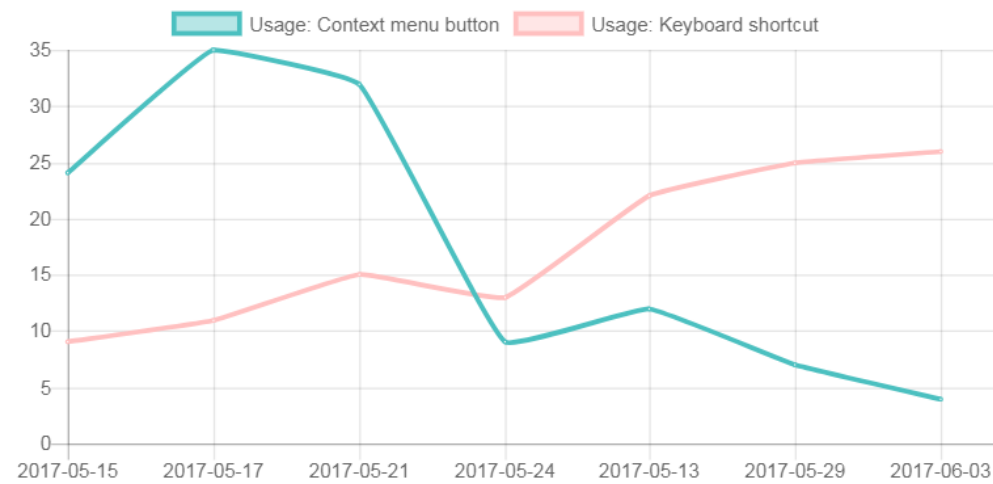


Figure 4.2: A feature utilisation metric collected from clients in a test scenario.

4.3 Publication III – Code quality measurement: case study

Background and objectives

Maintenance and upkeep are costly phases of the software life cycle. It has been estimated that maintenance can reach up to 92% of the total software costs (Kyte, 2012). Code quality can be analysed using various existing metrics, which can provide an estimate of the maintainability of software. There are several tools and frameworks that can be used for assessing the maintainability characteristics of a project. Many tools are included in IDEs, such as Eclipse metrics, JHawk and NDepend. As such, the existing tools are specific to the platform and programming language, providing quality analysis during development. Considering that maintenance also includes activities postrelease of a software product, it would be beneficial to perform quality measurement in the maintenance and upkeep phase of the life cycle.

In *Publication III*, we focused on the maintenance analysis of web applications. The focus on web applications provided a reasonably standardised interface for runtime performance through the browser's web API. We presented the design and implementation of a system called .Maintain (read: *dot maintain*). The .Maintain system included probes for gathering metrics in the system that were implemented in both the JavaScript and Ruby programming languages. The objective was to study how the .Maintain system could facilitate the systematic collection and analysis of maintenance metrics to reduce the effort required in the maintenance phase of software during development.

Results and contributions

In *Publication III*, we designed the architecture for a maintenance metrics collection and analysis system. As a result, we presented a tool for analysing and visualising the maintainability of a software project. The main contribution was the design, implementation, and evaluation of a system for collecting maintenance metrics. The design of the .Maintain tool is depicted in Figure 4.3, and the user interface is shown in Figure 4.4.

The novelty of the .Maintain system is the extendibility and modularity of architecture. This architecture is not platform specific. New probes and corresponding analysers can be added at any stage using the REST API with any programming language or platform. The data storage and reporting system provides a common interface for the systematic collection of quality metrics, allowing the developers of a project to establish and sustain a commitment for quality measurement.

In the context of the current study, the .Maintain system validates the ideas behind the measurement framework presented in *Publication II* by building a measurement platform for software quality issues. Providing a platform to establish measurement commitment is important because previous research has shown that the QA and testing practices of

developers do not necessarily line up with the measurement possibilities distinguished in academic research. For example, a recent study by Garousi and Felderer (2017) distinguishes that the industry and academia have different focus areas on software testing. Likewise, Antinyan et al. (2017) show that existing code complexity measures are poorly used in the industry. In the development of the .Maintain tool work, we used the maintainability index as an indicator of code quality because it has been used in both academia and industry. Thus, the main contribution of the present study is a tool that can be easily adopted by software developers.

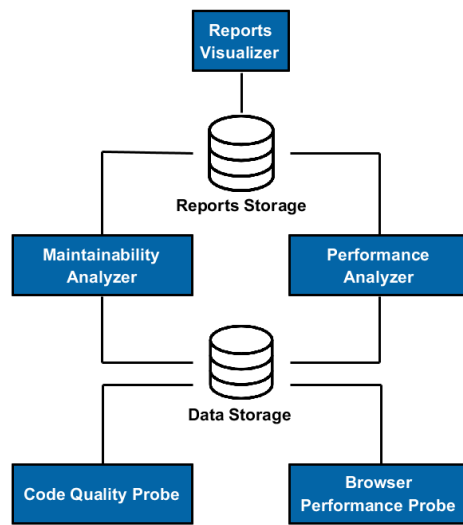


Figure 4.3: .Maintain system architecture.

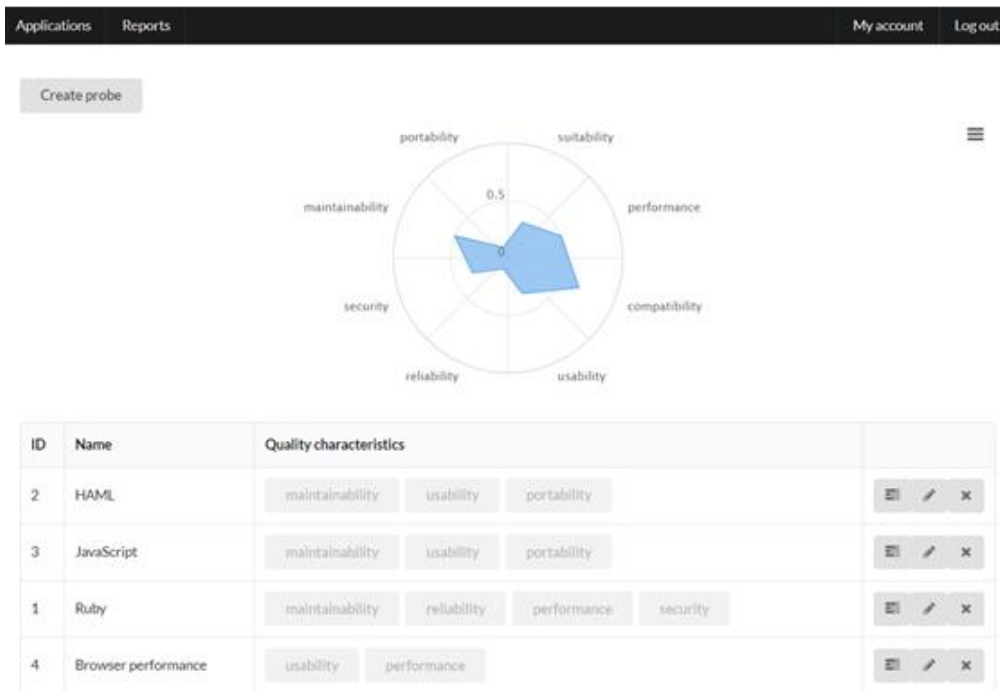


Figure 4.4: User interface for the .Maintain probes.

4.4 Publication IV – Early-warning system for software quality issues using maintenance metrics

Background and objectives

In the life cycle models for software, development is succeeded by an expensive and lengthy phase: maintenance. The growth of the maintenance phase and costs related to software maintenance work have been explored in numerous studies. There is no single reason for the trend, but there are several factors behind it, such as increasing complexity and integration of the systems (Banker et al., 1993), changing operation and operating environments of the systems (Reisman, 2006), the criticality of the systems (Capgemini, 2017) and the rise of a service-oriented approach to delivering software and their functionalities (Glass et al., 2006).

Many different approaches and technologies are aimed at reducing software maintenance costs, including, for example, SOA (MacKenzie et al., 2006), different delivery models (Humble & Farley, 2010), development and operations (DevOps; Ebert et al., 2016) and microservice architecture (Alshuqayran et al., 2016; Nadareishvili et al., 2016). These approaches are examples of improving software maintainability to reduce maintenance costs. However, the testing and deployment environments seem to be falling behind,

especially when it comes to the availability of generic tools because of the diversity of ecosystems.

On the other hand, there are techniques aimed at software quality estimation focusing on maintainability (Lewis & Henry, 1989), code metrics (Ferreira et al., 2012) or code smells (Fontana & Zaroni, 2011). These approaches can help identify problematic or defective parts of software systems. However, they require an interpretation because their key measurements are not compatible between projects.

Results and contributions

The objective of *Publication IV* was to further investigate the challenges of software maintenance. The research question was as follows: Is it possible to estimate the observed quality and maintenance needs of software using objective code metrics? A design science approach was used to implement a prototype of the .Maintain tool to calculate quality metrics based on the ISO/IEC 25010 quality attributes.

The development of the .Maintain tool was based on the principles of the quality characteristics, as defined in the ISO/IEC 25010 standard's quality models (ISO/IEC, 2011a), but here introducing two further steps. In the first step, measurement units called probes were integrated into the system during the development phase to assist in the data collection and activity logging work when a new feature was added during the maintenance work. Second, every time a new version of the system was deployed, the system analyses the quality outcomes from the data collected by the probes.

Based on the first prototype with three different commercial software projects, the basic premise of an early-warning system correlated with the project activity logs on the selected number of quality characteristics. Figure 4.5 shows an example graph of the analysis produced by the .Maintain tool. The findings showed that the maintenance indicators matched the code review and revision needs, indicating avenues for future development. In the context of the current study, *Publication IV* evaluated the .Maintain tool in large, real-world software construction projects. This extends the state-of-the-art by validating the utility of the tools designed in the previous publications.



Figure 4.5: Project quality report from the .Maintain toolkit, with annotated comments from the project manager interview(s).

4.5 Publication V – Guidelines for software testing education objectives from industry practices with a constructive alignment approach

Background and objectives

Software engineering educators can bridge the gap between formal education and industry practices to produce more industry-ready graduates by observing the industry in action. Good testing education can improve software quality; for example, students who are more experienced in testing may produce more reliable code. In this study, the objective was to align testing education content with industry practices.

To reach this objective, the data from *Publication I* were used to design learning objectives aligned with industry practices. The research questions for the study were as follows: 1) Which testing tools and technologies are most used in the industry? 2) What are the current issues related to testing in the industry? 3) How should the learning goals, teaching methods and evaluation methods in a software testing course be constructively aligned with current industry practices?

Results and contributions

The survey results on the testing practices in the industry were used to constructively align the software testing curriculum with industry practices and expectations, producing a course model responding to industry needs. The specific learning goals and activities are presented in Table 4.5. The model can be used as a frame of reference for learning objectives related to testing work in computer science education. Many suggestions for actual course content were presented.

Additionally, there are several guidelines for the better alignment of testing education and industry practices:

- Incorporate the use of the most common testing tools—defect reporting, unit testing and test automation—into the curriculum. The students will most likely require the skill to use these tools in their future workplaces.
- Use popular, widely used testing tools rather than the tools designed for education to teach students the correct use and configuration of real environments.
- Emphasise the importance of static testing methods as a way to improve code quality.
- Produce documentation early on to encourage a mindset for documenting the progress of the project.
- Use a variety of tools for the same purpose to give students the experience of the different tools available.
- Enforce documentation practices to enhance communication skills, for example, producing and handling defect reports.

Table 4.5: The constructive alignment of software testing course goals and methods to industry practices.

Learning goals	Teaching methods	Assessment methods (‘performances of understanding’)
Learn the practice of defect reporting and the use of bug tracking tools	Individual exercises: Find and report bugs	Demonstrate understanding through the individual projects
Implementing unit tests and evaluating test coverage	Individual exercises: Create a programme and set up unit tests	
Independent implementation of test automation	Individual exercises: Set up full testing automation for a programme	
Understand and apply test process design in future projects	Teamwork: Project management exercise and testing process simulation	Demonstrate understanding through equal contribution to the teamwork project (individual and group evaluation)
Integrating testing phases to software engineering practices	Teamwork: Project management exercise; acceptance testing between two teams	
Evaluating and managing technical debt; making rational compromises	Teacher-led exercise: A review of the shortcuts taken during the course, and discussion and evaluation of the long-term drawbacks of the shortcuts	Demonstrate understanding by a written assignment that reviews and evaluates technical issues
Implementing static testing: Creating checklists and performing code reviews	Teamwork: Going through checklists and reviewing each other’s code. TA acts as QA manager in final projects	Demonstrate understanding by working in a simulated verification and validation review

4.6 Publication VI – Designing early testing course curricula with activities matching the V-model phases

Background and objectives

Testing education improves software quality as testing-savvy students learn techniques that lead to more reliable programme code (Lemos et al., 2018). Previous research has established approaches to integrating testing and QA work into larger projects (Garousi, 2011; Krutz et al., 2014), but still, many institutions organise an undergraduate course in the methods and models of software testing separately. Perhaps, for this reason, students transitioning to the industry do not always have the necessary skills to test beforehand.

To place the testing activities in a software engineering context, we contrasted them with the phases in the V-model (Mathur & Malik, 2010; Rook, 1986). The V-model is a generic software development process model in which requirement analysis, specification, architectural design, and detail design are linked with the levels of testing (acceptance testing, system testing, integration testing, and unit testing). These development process phases and testing levels are often referenced in software engineering education. However, in the context of education and training, the practical impact of these activities may play an auxiliary role or even be neglected. Hence, students might be familiar with the development process phases on an abstract level but fail to understand which practical activities should happen within them.

In this study, the objective was to investigate the following: 1) What learning activities can we map to the high-level testing levels? 2) Which actual testing techniques can be utilised? 3) How do those activities and high-level concepts relate to other software engineering processes? To answer these questions, we designed an undergraduate course on the fundamentals of software testing, here with a specific focus on the V-model phases and concrete testing activities.

Results and contributions

In the study, an introductory software testing course was designed and using the principles of constructive alignment, and learning goals were mapped to weekly activities and testing techniques. The course structure is presented in Table 4.6. The study evaluated the course structure by examining student outcomes. Students' practical assignments were used as demonstrations of learning.

From the projects, we observed that students were able to adopt the testing mindset and carry out comprehensive and systematic testing at the system testing level. On the other hand, this systematic approach to testing work was mainly carried out at the system level, while many projects had problems with unit tests, integration tests, and reporting of the project.

Table 4.6: The constructive alignment of software testing course goals and methods to industry practices.

Week	Development phase (V-model)	Test level (V-model)	Weekly topic(s)	Activities and testing techniques applied	Learning goals
1	Specification and requirement analysis	System testing	Introduction to testing Objectives of testing	Black-box system testing. Exploratory testing. Boundary value analysis. Defect reporting.	Understand the objectives of testing work. The students can create a (black-box) test cases. The students understand the scope, and limitations of the black-box methods.
2	Detail design	Unit testing	Testing levels Unit testing	White-box testing. Test case reporting. Equivalence partitioning.	Understand the concept of unit/module test. The students understand the difference between black-box and white-box testing.
3	Architectural design	Integration testing	Integration testing	Combinatorial methods and the classification tree method. Test stubs.	Understand the infeasibility of ‘testing everything’. The students can select a technique for deriving test cases. The students understand the scope and limitations of the software testing in real-world software projects.
4	Specification and requirement analysis	System testing Acceptance testing	System testing	State transition testing. Scenario testing. Random testing.	Understand the objectives of system-level testing. The students can select an appropriate testing technique for system testing. The students understand the scope and limitations of the system-level testing methods.

Table 4.6 (continued).

Week	Development phase (V-model)	Test level (V-model)	Weekly topic(s)	Activities and testing techniques applied	Learning goals
5	Detail design	Unit testing	Test automation and tools	Implementing unit tests in code, using a unit testing framework.	The students can use a programming framework/library to implement module tests. The student understands the scope and limitations of the unit testing tools.
6	Architectural design	System testing	Testing processes, documentation and planning	Creating test plans. Code review and static testing methods. Test coverage analysis.	The students understand the purpose of static testing methods and code review practices.
7	Specification, architectural design, detail design	System, integration and unit testing	Visiting lecture from a software company	Course project: Plan, design, implement and document testing for a small software item.	The students can demonstrate their knowledge by applying the course's activities autonomously in the testing project. The students can explain how test process activities would relate to the whole software project.

4.7 Summary of contributions

The publications in the current thesis provide the following contributions to the state of the art in software testing and QA.

Understanding software testing practices in the industry. The survey conducted in *Publication I* explored current practices related to testing and QA in the Finnish software industry. Additionally, the survey revealed changes in practices within the past few years. According to the survey results, the organisations have shifted towards test automation and more sophisticated testing infrastructure, they apply more agile practices even in mission-critical software, and they have reduced the use of formal process models.

The growing use of automation and tools have shifted development practices towards more agile and less formal methods, highlighting the need for better, more intelligent automated tests and QA tools. The reduced use of formal processes and need to push new features into products translate into the need for better support for acceptance testing, regression testing and QA in general. This served as motivation for the research in the subsequent publications.

A framework for measuring maintenance needs using runtime metrics. In *Publication II*, a framework was designed for the runtime measurement of maintenance needs. This framework can be used to design, implement and sustain the commitment for quality measurement during the software development process. The framework also provides actionable suggestions for how to measure different quality characteristics using runtime probes and code quality metrics in software. In the subsequent publications, the framework was used as a roadmap for building the .Maintain tool, which implements runtime metrics and code quality measurements in practice.

Implementing tools for measuring maintenance needs in practice. *Publications II* and *III* utilised the design science approach to design, construct, evaluate and validate the .Maintain tool for measuring software quality characteristics. Through the design and implementation of proof-of-concept prototypes and working software artefacts, we demonstrated that the .Maintain tool can be used as an early-warning system for detecting quality issues. The utility of the tool was demonstrated by using real-world software projects and mature products already in their maintenance phase.

A curriculum, activities and learning objectives for testing education and guidelines for aligning the learning objectives with industry practices. *Publications V* and *VI* presented an exploration of the pedagogical practices of testing education. Starting with the survey results presented in *Publication I*, a testing curriculum and guidelines for better aligning the learning objectives with the current industry practices were constructed.

The presented course model incorporates industry practices and expectations into a testing course curriculum. Learning goals, teaching methods and assessment methods in addition to the different knowledge units were constructively aligned with the surveyed practices. Because the results presented in *Publication V* and *Publication VI* are concerned with learning objectives and pedagogical guidelines rather than specific tools or technologies, they can be used in many different contexts. Therefore, the results are valuable to a wide range of educators.

5 Discussion

This chapter summarises the objectives, methods and contributions presented in the current thesis. First, the objectives and methods are revisited. The research questions are then answered. Finally, the validity of the results is assessed, and future research avenues are presented.

5.1 Research objectives

The objective of the present thesis was to investigate development directions related to software testing and QA work. The study began by conducting a survey to establish the state-of-the-art in current testing practices. Next, novel tools for measuring software quality and detecting maintenance issues were explored. Finally, testing education was investigated to better prepare students for software engineering work in the industry.

In *Publication I*, the survey method (Fink & Kosecoff, 1985) was used to elicit views on testing and QA practices. People working in the software industry were asked to participate in the survey, and we collected responses from different companies at the organisational unit level. The objective was to explore industry practices concerning software testing.

Publications II, III and IV employed the DSR method (Hevner et al., 2004; Peffers et al., 2007). The framework for observing maintenance needs and the .Maintain tool were designed for runtime metrics collection of software projects. The .Maintain tool implemented the framework in practice. .Maintain was used in several real software projects, and the evaluation of the tool acted as a proof of concept.

Publications V and VI investigated education and training related to testing and QA work. The industry practices (uncovered in *Publication I*) were mapped to learning activities, learning objectives and practical testing techniques to form an industry-aligned testing curriculum. These studies employed the constructive alignment research method (Biggs, 1996, 2014).

5.2 Findings

Next, we address the research questions individually and present the main contributions of the current thesis. The following text synthesises the contributions of *Publications I–VI* in the context of the research questions.

RQ 1. What is the current state of the industry practices in testing and QA, and how have they evolved in recent times?

The data in *Publication I* revealed changing practices in the industry within the past few years. Organisations have shifted towards test automation and a more sophisticated

testing infrastructure, have applied more agile practices even in mission-critical software and have reduced the use of formal process models.

The most popular tools used include defect reporting tools, test automation tools and unit testing tools. The configurability of testing tools has become an issue, and support for different software platforms might become an issue when observing a trend in the changes. Additionally, feature development during late development phases shortens testing schedules.

RQ 2. What kind of framework would enable measurement of software quality characteristics and detecting maintenance issues?

In *Publication II*, a framework for collecting runtime metrics was proposed as one solution for the growing maintenance costs. Measurement probes were linked into the software during the development phase and used to collect quality information during the runtime. As a proof-of-concept, the measurements were implemented in an open-source software project. Examples of useful scenarios were presented to demonstrate the utility of the framework.

RQ 3. To what extent can runtime quality metrics be collected from real software projects to analyse quality and maintainability?

Publications III and *IV* further demonstrated the idea of measuring runtime software metrics. As a result, the .Maintain tool for analysing and visualising the maintainability of a software project was presented. The results of the studies showed that the maintenance indicators matched the code review and revision needs, indicating further avenues for future development.

The novelty of the .Maintain tool is the extendibility and modularity of its architecture. The .Maintain architecture is not platform specific. Instead, new probes and corresponding analysers can be added at any stage using the REST API, with any programming language or platform. The presented studies showed that the tool can be used to estimate project quality and provide an early warning of issues that may arise.

RQ 4. To what extent are software engineers ready to use the testing and QA tools, and how can testing education be better oriented to support this goal?

In *Publications V* and *VI*, the education and training in testing for software professionals were investigated. We observed that students could adopt the testing mindset and carry out comprehensive and systematic testing at the system test level. However, the fact that the systematic approach to testing work was mainly carried out at the system level could be seen as a problem because many students had problems with unit tests, integration tests and reporting.

The principles of constructive alignment were used to develop learning activities, learning goals, teaching methods and assessment methods aligning with the industry requirements.

Concrete learning objectives were created using common software engineering methods and models. This helped better frame the testing topics for software developers.

Main RQ. To what extent can test automation and software measurement tools improve testing and QA work in software companies?

Finally, to answer the main research question, the software industry has exhibited a drive towards more automation and agile practices. At the same time, the testing and deployment environments seem to be falling behind the rate of development, making QA work more challenging to automate. New, smart tools in software development can help alleviate this disparity. In the current thesis, the .Maintain tool was presented as one solution to the growing need for automation in QA. The .Maintain tool made it possible to detect changes in the software quality during development. This can help identify defects or high-maintenance modules in the software. Additionally, the curriculum for testing education and training can help quickly bring new software developers up to speed with industry practices. With more knowledge of testing, the software engineering workforce will be better equipped to perform QA activities, and thus be better prepared to use automation and measurement tools.

5.3 Implications for practice and research

Explorations into the measurement of software defects and maintenance needs:

Previous research has shown that the QA and testing practices of developers are not in line with the measurement possibilities distinguished in academic research. Existing code complexity measures are poorly used in the industry. In fact, industry and academia have completely different focus areas on software testing related topics. The research avenues related to the measurement and monitoring of software products are fruitful. This was further demonstrated in the evaluation of the .Maintain tool, which suggested that there is a need for further study and refinement in the development of software quality measurement monitoring systems.

Further understanding about software defects in agile development: Surveying the software industry revealed changing practices. The software industry has increasingly employed tools to support software development. Organisations rely heavily on automation and employ agile practices. However, these tools are also the cause of many configuration problems. The need to push new features means that the products need better support for acceptance testing, regression testing and, in general, better QA. This suggests that more research is needed to understand how and why software defects emerge in the agile development process.

Design a testing curriculum for software engineering: The processes and tools used in the industry can be challenging to teach because of the sheer number of different tools available and how different companies may employ slightly different ways to utilise them. Therefore, more research is needed in designing the testing curriculum for software engineering. Students seem to grasp some QA-related topics instinctively, while other

topics proved more challenging to teach. This might sway the learning outcomes of testing education towards certain topics more than educators intend.

5.4 Assessment of the research

The limitations and quality of the current study warrant discussion. This section addresses the quality and limitations of the study through the lens of reliability and validity based on the recommendations of Wohlin et al. (2012) and Yin (2009, 2011). In particular, the research programme is assessed in terms of its reliability, construct validity, internal validity and external validity.

The quality of research can be expressed through the concepts of *reliability* and *validity*. Reliability is the degree to which the results of a study are replicable (Dubois & Gibbert, 2010). Validity is often broken down into smaller measures, all of which indicate the consistency between study protocols and results.

Construct validity is a measure of the degree to which the research instruments are in line with the findings, that is, how accurate the conclusions are, while also asking if the study has investigated what it claims to have investigated in the research questions (Dubois & Gibbert, 2010). *Internal validity* (Dubois & Gibbert, 2010; Yin, 2009) is the measure of the consistency between data and the interpretations made of it, that is, how well the study establishes cause and effect. Finally, *external validity* (Lavrakas, 2008; Yin, 2009) is a measure of generalizability for study results.

Reliability

In Phase 1 of the research programme, a survey was conducted following the method by Fink and Kosecoff (1985). Kitchenham et al. (2002) divide survey studies into exploratory studies, from which estimates can be drawn, and confirmatory studies, from which strong conclusions can be drawn. In the context of this work, the survey is considered an exploratory, observational and cross-sectional study exploring testing practices in the industry. *Publication I* documented the survey instrument and results. The survey design and anonymised data are also available in an online repository (Hynninen et al., 2017).

In Phases 2 and 3, the work employed a design science approach. Unlike traditional qualitative research methods, DSR involves a degree of creativity. Thus, DSR is not always easily replicable, which conflicts with the objective of reliability (Kuechler & Vaishnavi, 2011). However, the work presented in *Publications II, III* and *IV* follows an iterative approach to improving the outcomes at each round. The work started by designing a measurement framework, whose utility was demonstrated by using use cases and descriptive scenarios. The work continued with the development of the measurement framework and then continued by implementing the tools to realise the measurement of quality attributes and maintenance needs in practice. Thus, the .Maintain tool was created

as a prototype, and it was consecutively put into use in a realistic environment, which further demonstrated the utility and novelty of the work.

In Phase 4, we investigated the organisation and content of a testing curriculum. The work focused mainly on studying the learning outcomes of students during one semester, and the studies yielded no quantitative results. However, the research approach in this phase was, again, more qualitative, focusing on whether the students completed the individual learning objectives instead of analysing descriptive statistics like course grades or student evaluations of teaching.

Construct validity

In Phase 1, the survey employed questions from a survey instrument that was validated in prior studies (e.g., Kasurinen et al., 2010). The survey instrument was developed over the years and used in multiple studies.

In Phases 2 and 3, the software quality measurement framework and the .Maintain tool were designed to implement the ISO 25010 (ISO/IEC, 2011b) software quality model(s). This approach was similar to many prior studies, in which software quality measurement frameworks or tools had been constructed.

In Phase 4, the design of the curriculum was based on the ACM/IEEE guidelines for degree programmes in software engineering (Ardis et al., 2015). In addition, learning activities and objectives were derived from the empirical results collected in the survey in Phase 1.

Internal validity

In Phase 1, the survey instrument was based on prior studies. The survey contained multiple-item, multiple-choice and open-ended questions. Previous studies used Cronbach's alpha (Cronbach, 1951) as a validity test for the survey questions: Cronbach's alpha expresses the degree to which items in a scale are homogeneous (Cho, 2016; Cronbach, 1951). The survey data were also compared with prior studies, which facilitated the observation of changes in practices. A team of four researchers was involved, which further facilitated the triangulation (Denzin, 1973) of the findings.

In Phases 2 and 3, the research revolved around testing the .Maintain tool and prototypes leading up to the tool's creation. In the work, we used creativity and experience in the field to create plausible test scenarios that would be useful in real-world software development projects. The results were verified using a case study in a realistic environment. When the tool was used in a real environment, the researchers had access to developers and their version control data.

In Phase 4, the objective was to evaluate the testing and QA competencies that software engineers receive in their education. The work was carried out over one semester, meaning that the longitudinal effects of the curriculum require future evaluation. In

particular, the results regarding the success of the alignment between learning objectives and industry practices are still preliminary.

External validity

In Phase 1, the survey sample was geographically limited to Finnish software companies. It is possible that this sample was not representative of companies in different countries or different socioeconomic contexts. However, the results are in line with similar surveys around the world, suggesting that the results can be generalisable. The response rate was comparable with other online surveys, and the observations were presented as explorative, not as strong conclusions.

In Phases 2 and 3, the testing of the .Maintain tool was done in a real-world environment. The evaluation of the tool was done only once because of time and resource constraints, hence posing a possible threat to validity. However, many prior studies have used the same starting point to conduct similar research efforts. In addition, the findings were generally in line with the literature.

In Phase 4, the studies investigated testing education in the Finnish context. However, the results are considered exploratory, from which estimates can be drawn. Additionally, the presented guidelines and learning objectives should be universal because they mirror knowledge areas in universally known recommendations, such as SWEBOK (Bourque et al., 1999) and the V-model (Mathur & Malik, 2010).

6 Conclusion

The objective of the current thesis was to investigate development directions in testing and QA. The main research question—*to what extent can automation and tools improve testing and QA work in software companies*—was answered in four phases. First, a survey was conducted to map current practices related to testing and QA in the Finnish software industry. Next, a framework for continuous software measurement was investigated. This framework was utilised when designing and implementing the .Maintain tool, whose utility as an early-warning system for software defects during development and maintenance was demonstrated in practice. Finally, the present thesis investigated the aspects of education and training in the field of testing.

Surveying the software industry has revealed changing practices that have taken place over the past eight years. Organisations have been relying more on testing automation and employing more agile practices. Tools are no longer seen as limited in terms of their functionality, but at the same time, configuration problems and more complex platforms have become more common.

To reduce the complexity related to the monitoring of quality aspects, the current thesis proceeded to propose a framework and, consequently, a tool for measuring quality characteristics in software development projects. The .Maintain tool was demonstrated to be a useful early-warning system for quality-related issues because the issues indicated by the tool matched code review findings and expert evaluation. The evaluation of the .Maintain tool provided evidence that the automatic measurement of software project characteristics is an interesting avenue of research, which is in line with findings in recent related research.

Finally, the current study investigated the capabilities of computer and software engineering students in software testing. The students tended to have a curious and rigorous mindset but only regarding certain areas of testing, for example, system testing. Additionally, there are many tools used in the industry, and providing a holistic learning experience about testing can be challenging for educators.

In short, the software industry has taken an increasing number of tools to support both the software development process and QA work. The design and development of better tools is a contemporary research topic. The need and utility for collecting quality metrics and maintenance needs was demonstrated with the .Maintain tool. However, the processes and tools used in the industry can be challenging to teach, and more research is needed to design the curriculum for competent developers.

The main contributions of the current thesis are threefold. First, the current study contributes to the knowledge of software testing practices in the industry. Second, the framework for measuring maintenance needs using runtime metrics and the .Maintain tool for demonstrating this approach were constructed. Third, a curriculum, learning

activities and learning objectives for testing education were presented, in addition to guidelines for aligning the learning objectives with industry practices.

The software industry is rapidly moving towards automation, which has become a standard in everyday software development work. There is an ever-growing need to push new features into products, which, in turn, calls for better acceptance testing, regression testing, late testing and QA work in the publishing, deployment and maintenance phases.

The drive towards automation is not without problems, however. For example, as discovered in the first phase of the current research programme, automation is seen as a difficult feat because of its low reuse possibilities and high cost. Additionally, as the work and tools become more complex, there are more misunderstandings between developers and testers, in turn slowing down the rate of development. However, despite these obstacles, companies are moving towards automation and more agile practices, shifting away from plan-based, formal development processes. The tools and processes used play a vital role in this trend.

References

- Alshuqayran, N., Ali, N., & Evans, R. (2016). A systematic mapping study in microservice architecture. *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, 44–51.
- Antinyan, V., Staron, M., & Sandberg, A. (2017). Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time. *Empirical Software Engineering*, 1–31.
- Ardis, M., Budgen, D., Hislop, G. W., Offutt, J., Sebern, M., & Visser, W. (2015). SE 2014: Curriculum guidelines for undergraduate degree programs in software engineering. *Computer*, 11, 106–109.
- Bai, G. R., Presler-Marshall, K., Price, T. W., & Stolee, K. T. (2022). Check It Off: Exploring the Impact of a Checklist Intervention on the Quality of Student-authored Unit Tests. *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1*, 276–282.
- Bai, G. R., Smith, J., & Stolee, K. T. (2021). How Students Unit Test: Perceptions, Practices, and Pitfalls. *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, 248–254. <https://doi.org/10.1145/3430665.3456368>
- Banker, R. D., Datar, S. M., Kemerer, C. F., & Zweig, D. (1993). Software Complexity and Maintenance Costs. *Commun. ACM*, 36(11), 81–94. <https://doi.org/10.1145/163359.163375>
- Bautista, L., Abran, A., & April, A. (2012). Design of a performance measurement framework for cloud computing. *Journal of Software Engineering and Applications*, 5(02), 69.
- Biggs, J. (1996). Enhancing teaching through constructive alignment. *Higher Education*, 32(3), 347–364.
- Biggs, J. (2014). Constructive alignment in university teaching. *HERDSA Review of Higher Education*, 1(5), 5–22.
- Bourque, P., Dupuis, R., Abran, A., Moore, J. W., & Tripp, L. (1999). The guide to the software engineering body of knowledge. *IEEE Software*, 16(6), 35–44.
- Britton, T., Jeng, L., Carver, G., Cheak, P., & Katzenellenbogen, T. (2013). Reversible debugging software. *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep.*

- Capgemini. (2017). *World Quality Report 2016-17*. Capgemini Group. <https://www.capgemini.com/thought-leadership/world-quality-report-2016-17>
- Carstensen, A.-K., & Bernhard, J. (2019). Design science research – a powerful tool for improving methods in engineering education research. *European Journal of Engineering Education*, 44(1–2), 85–102. <https://doi.org/10.1080/03043797.2018.1498459>
- Chen, L. (2015). Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, 32(2), 50–54.
- Cho, E. (2016). Making Reliability Reliable: A Systematic Approach to Reliability Coefficients. *Organizational Research Methods*, 19(4), 651–682. <https://doi.org/10.1177/1094428116656239>
- Cronbach, L. J. (1951). Coefficient alpha and the internal structure of tests. *Psychometrika*, 16(3), 297–334. <https://doi.org/10.1007/BF02310555>
- Czerwonka, J., Nagappan, N., Schulte, W., & Murphy, B. (2013). CODEMINE: Building a Software Development Data Analytics Platform at Microsoft. *IEEE Software*, 30(4), 64–71. <https://doi.org/10.1109/MS.2013.68>
- Denzin, N. K. (1973). *The research act: A theoretical introduction to sociological methods*. Transaction publishers.
- Desai, C., Janzen, D., & Savage, K. (2008). A survey of evidence for test-driven development in academia. *ACM SIGCSE Bulletin*, 40(2), 97–101.
- Dubois, A., & Gibbert, M. (2010). From complexity to transparency: Managing the interplay between theory, method and empirical phenomena in IMM case studies. *Industrial Marketing Management*, 39(1), 129–136.
- Ebert, C., Gallardo, G., Hernantes, J., & Serrano, N. (2016). DevOps. *Ieee Software*, 33(3), 94–100.
- Ferreira, K. A., Bigonha, M. A., Bigonha, R. S., Mendes, L. F., & Almeida, H. C. (2012). Identifying thresholds for object-oriented software metrics. *Journal of Systems and Software*, 85(2), 244–257.
- Fink, A., & Kosecoff, J. (1985). *How to conduct surveys: A step-by-step guide*. Sage Publications.
- Fischer, C., Winter, R., & Wortmann, F. (2010). Design Theory. *Business & Information Systems Engineering*, 2(6), 387–390. <https://doi.org/10.1007/s12599-010-0128-2>

- Fontana, F. A., & Zanoni, M. (2011). On investigating code smells correlations. *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, 474–475.
- Garousi, V. (2011). Incorporating real-world industrial testing projects in software testing courses: Opportunities, challenges, and lessons learned. *2011 24th IEEE-CS Conference on Software Engineering Education and Training (CSEE&T)*, 396–400.
- Garousi, V., Arkan, S., Urul, G., Karapıçak, Ç. M., & Felderer, M. (2020). Assessing the maturity of software testing services using CMMI-SVC: An industrial case study. *ArXiv Preprint ArXiv:2005.12570*.
- Garousi, V., & Felderer, M. (2017). Worlds Apart: Industrial and Academic Focus Areas in Software Testing. *IEEE Software*, 34(5), 38–45. <https://doi.org/10.1109/MS.2017.3641116>
- Garousi, V., Rainer, A., Lauvås Jr, P., & Arcuri, A. (2020). Software-testing education: A systematic literature mapping. *Journal of Systems and Software*, 165, 110570.
- Glass, R. L., Collard, R., Bertolino, A., Bach, J., & Kaner, C. (2006). Software testing and industry needs. *IEEE Software*, 23(4), 55–57.
- Gregor, S., & Hevner, A. R. (2013). Positioning and presenting design science research for maximum impact. *MIS Quarterly*, 37(2), 337–355.
- Hegedus, P. (2013). Revealing the Effect of Coding Practices on Software Maintainability. *2013 29th IEEE International Conference on Software Maintenance (ICSM)*, 578–581. <https://doi.org/10.1109/ICSM.2013.99>
- Hevner, A., & Chatterjee, S. (2010). Design science research frameworks. In *Design research in information systems* (pp. 23–31). Springer.
- Hevner, A. R. (2007). A three cycle view of design science research. *Scandinavian Journal of Information Systems*, 19(2), 4.
- Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design Science in Information Systems Research. *MIS Q.*, 28(1), 75–105.
- Humble, J., & Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education.
- Hynninen, T., Kasurinen, J., Knutas, A., & Taipale, O. (2017). *Survey Data for “Software Testing: Survey of the Industry Practices”* [Data set]. Zenodo. <https://doi.org/10.5281/zenodo.803996>

- IEEE. (2011). IEEE Draft Guide: Adoption of the Project Management Institute (PMI) Standard: A Guide to the Project Management Body of Knowledge (PMBOK Guide)-2008 (4th edition). *IEEE P1490/D1*, May 2011, 1–505. <https://doi.org/10.1109/IEEESTD.2011.5937011>
- ISO/IEC. (2005). *ISO/IEC 9000 Family: Quality management*.
- ISO/IEC. (2006). *ISO/IEC 14764: Software Engineering—Software Life Cycle Processes—Maintenance*.
- ISO/IEC. (2011a). *ISO/IEC 25000: Systems and software Quality Requirements and Evaluation (SQuaRE)—Guide to SQuaRE*.
- ISO/IEC. (2011b). *ISO/IEC 25010: Systems and software Quality Requirements and Evaluation (SQuaRE)—System and software quality models*.
- ISO/IEC. (2013). *ISO/IEC 29119-1 Software and systems engineering—Software testing—Part 1: Concepts and definitions*.
- ISO/IEC. (2017). *ISO/IEC 24765: Systems and software engineering—Vocabulary*.
- Janus, A., Schmietendorf, A., Dumke, R., & Jäger, J. (2012). The 3C Approach for Agile Quality Assurance. *Proceedings of the 3rd International Workshop on Emerging Trends in Software Metrics*, 9–13. <http://dl.acm.org/citation.cfm?id=2669379.2669382>
- Kaner, C. (2006, November 17). *Exploratory Testing* [Keynote]. Quality Assurance Institute Worldwide Annual Software Testing Conference, Orlando, FL, USA.
- Kasurinen, J., Maglyas, A., & Smolander, K. (2014). Is Requirements Engineering Useless in Game Development? *Requirements Engineering: Foundation for Software Quality*, 1–16. https://doi.org/10.1007/978-3-319-05843-6_1
- Kasurinen, J. P. (2013). Ohjelmistotestauksen käsikirja. Jyväskylä: *Docendo, 1*.
- Kasurinen, J., Taipale, O., & Smolander, K. (2010). Software test automation in practice: Empirical observations. *Advances in Software Engineering*, 2010. <https://www.hindawi.com/journals/ase/2010/620836/abs/>
- Kit, E. (1995). *Software testing in the real world*. Addison-wesley. <http://cds.cern.ch/record/362136>
- Kitchenham, B. A., Pfleeger, S. L., Pickard, L. M., Jones, P. W., Hoaglin, D. C., El Emam, K., & Rosenberg, J. (2002). Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8), 721–734.

- Krutz, D. E., Malachowsky, S. A., & Reichlmayr, T. (2014). Using a real world project in a software testing course. *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, 49–54.
- Kuechler, B., & Vaishnavi, V. (2008). On theory development in design science research: Anatomy of a research project. *European Journal of Information Systems*, 17(5), 489–504. <https://doi.org/10.1057/ejis.2008.40>
- Kuechler, W., & Vaishnavi, V. (2011). Promoting relevance in IS research: An informing system for design science research. *Informing Science*, 14, 125.
- Kyte, A. (2012). *The four laws of application, total cost of ownership*. Gartner, Inc.
- Lauvås Jr, P., & Arcuri, A. (2018). Recent trends in software testing education: A systematic literature review. *Norsk IKT-Konferanse for Forskning Og Utdanning*.
- Lavrakas, P. J. (2008). *Encyclopedia of survey research methods*. Sage publications.
- Lazzarini Lemos, O. A., Fagundes Silveira, F., Cutigi Ferrari, F., & Garcia, A. (2017). The impact of Software Testing education on code reliability: An empirical assessment. *Journal of Systems and Software*. <https://doi.org/10.1016/j.jss.2017.02.042>
- Lazzarini Lemos, O. A. L., Cutigi Ferrari, C., Fagundes Silveira, F., & Garcia, A. (2015). Experience report: Can software testing education lead to more reliable code? *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, 359–369. <https://doi.org/10.1109/ISSRE.2015.7381829>
- Lemos, O. A. L., Silveira, F. F., Ferrari, F. C., & Garcia, A. (2018). The impact of Software Testing education on code reliability: An empirical assessment. *Journal of Systems and Software*, 137, 497–511.
- Leppänen, M., Mäkinen, S., Pagels, M., Eloranta, V.-P., Itkonen, J., Mäntylä, M. V., & Männistö, T. (2015). The highways and country roads to continuous deployment. *Ieee Software*, 32(2), 64–72.
- Lewis, J., & Henry, S. (1989). A methodology for integrating maintainability using software metrics. , *Conference on Software Maintenance, 1989., Proceedings*, 32–39. <https://doi.org/10.1109/ICSM.1989.65191>
- Lopez, G., Coccozza, F., Martinez, A., & Jenkins, M. (2015). Design and implementation of a software testing training course. *2015 ASEE Annual Conference & Exposition*, 26–453.
- Ma, D. (2007). The business model of" software-as-a-service". *Ieee International Conference on Services Computing (Scc 2007)*, 701–702.

- MacKenzie, C. M., Laskey, K., McCabe, F., Brown, P. F., Metz, R., & Hamilton, B. A. (2006). Reference model for service oriented architecture 1.0. *OASIS Standard*, 12(S 18).
- March, S. T., & Smith, G. F. (1995). Design and natural science research on information technology. *Decision Support Systems*, 15(4), 251–266.
- March, S. T., & Storey, V. C. (2008). Design science in the information systems discipline: An introduction to the special issue on design science research. *MIS Quarterly*, 725–730.
- Mathur, S., & Malik, S. (2010). Advancements in the V-Model. *International Journal of Computer Applications*, 1(12), 29–34.
- Molnar, A.-J., & Motogna, S. (2020). Longitudinal Evaluation of Open-Source Software Maintainability. *ArXiv Preprint ArXiv:2003.00447*.
- Motogna, S., Vescan, A., Serban, C., & Tirban, P. (2016). An approach to assess maintainability change. *2016 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, 1–6. <https://doi.org/10.1109/AQTR.2016.7501279>
- Myers, G. J., Badgett, T., Thomas, T. M., & Sandler, C. (2004). *The art of software testing* (Vol. 2). Wiley Online Library.
- Nadareishvili, I., Mitra, R., McLarty, M., & Amundsen, M. (2016). *Microservice architecture: Aligning principles, practices, and culture*. O'Reilly Media, Inc.
- Osterweil, L. (1996). Strategic directions in software quality. *ACM Computing Surveys (CSUR)*, 28(4), 738–750.
- Peppers, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of Management Information Systems*, 24(3), 45–77.
- Planning, S. (2002). *The economic impacts of inadequate infrastructure for software testing*. <https://www.nist.gov/document/report02-3pdf>
- Prasad, L., Yadav, R., & Vore, N. (2021). A Systematic Literature Review of Automated Software Testing Tool. *Proceedings of 3rd International Conference on Computing Informatics and Networks: ICCIN 2020*, 101–123.
- Reisman, S. (2006). Costs and benefits of software engineering in product development environments. In *Cases on Strategic Information Systems* (pp. 199–215). IGI Global.

- Rook, P. (1986). Controlling software projects. *Software Engineering Journal*, 1(1), 7–16.
- Sarwar, M. I., Tanveer, W., Sarwar, I., & Mahmood, W. (2008). A comparative study of MI tools: Defining the Roadmap to MI tools standardization. *Multitopic Conference, 2008. INMIC 2008. IEEE International*, 379–385. <https://doi.org/10.1109/INMIC.2008.4777767>
- Scatalon, L. P., Carver, J. C., Garcia, R. E., & Barbosa, E. F. (2019). Software testing in introductory programming courses: A systematic mapping study. *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 421–427.
- Scatalon, L. P., Prates, J. M., De Souza, D. M., Barbosa, E. F., & Garcia, R. E. (2017). Towards the role of test design in programming assignments. *2017 IEEE 30th Conference on Software Engineering Education and Training (CSEE&T)*, 170–179.
- Shaughnessy, J. J., Zechmeister, E. B., & Zechmeister, J. S. (2012). *Research methods in psychology* (9th ed). McGraw-Hill.
- Taipale, O., & Smolander, K. (2006). Improving software testing by observing practice. *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, 262–271.
- Taipale, O., Smolander, K., & Kälviäinen, H. (2005). Finding and Ranking Research Directions for Software Testing. In I. Richardson, P. Abrahamsson, & R. Messnarz (Eds.), *Software Process Improvement: 12th European Conference, EuroSPI 2005, Budapest, Hungary, November 9-11, 2005. Proceedings* (pp. 39–48). Springer Berlin Heidelberg. https://doi.org/10.1007/11586012_5
- Tassey, G. (2002). The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*.
- Valle, P. H. D., Toda, A. M., Barbosa, E. F., & Maldonado, J. C. (2017). Educational games: A contribution to software testing education. *2017 IEEE Frontiers in Education Conference (FIE)*, 1–8.
- Wang, Y., Mäntylä, M., Demeyer, S., Wiklund, K., Eldh, S., & Kairi, T. (2020). Software Test Automation Maturity: A Survey of the State of the Practice. *Proceedings of the 15th International Conference on Software Technologies*, 27–38. <https://doi.org/10.5220/0009766800270038>
- Whittaker, J. A. (2000). What is software testing? And why is it so hard? *IEEE Software*, 17(1), 70–79.

- Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). *Experimentation in software engineering*. Springer Science & Business Media.
- Yamashita, A. (2015). Experiences from performing software quality evaluations via combining benchmark-based metrics analysis, software visualization, and expert assessment. *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 421–428. <https://doi.org/10.1109/ICSM.2015.7332493>
- Yin, R. K. (2009). *Case study research: Design and methods* (Vol. 5). Sage Publications.
- Yin, R. K. (2011). *Applications of case study research*. Sage Publications.

Publication I

Hynninen, T., Kasurinen, J., Knutas, A., and Taipale, O.
Software testing: Survey of the industry practices

In Proceedings of the 2018 41st International Convention on Information and
Communication Technology, Electronics and Microelectronics (MIPRO)
Pp. 1449-1454, 2018
© 2018, IEEE

Software Testing: Survey of the Industry Practices

T. Hynninen*, J. Kasurinen**, A. Knutas*** and O. Taipale*

* Lappeenranta University of Technology, Lappeenranta, Finland

** South-Eastern Finland University of Applied Sciences (XAMK), Kotka, Finland

*** Lero, the Irish Software Research Centre, Dublin, Ireland

timo.hynninen@lut.fi, jussi.kasurinen@xamk.fi, antti.knutas@lero.ie, ossi.taipale@lut.fi

Abstract - The objective of this survey was to explore industry practices concerning software testing. We studied software organizations to assess how they test their products and what process models they follow. The data collection was implemented as an online implementation of the survey method. Additionally the collected data was compared to our prior survey study to understand how the industry practices have changed. According to our results, the organizations have shifted towards test automation and more sophisticated testing infrastructure, they apply more agile practices even in the mission-critical software, and they have reduced the use of formal process models.

Keywords - *software testing; survey; industry practices; quality assurance*

I. INTRODUCTION

Testing can be one of the most expensive tasks for any software project. Besides causing immediate costs, problems of testing are also related to the costs of poor quality, malfunctioning programs and errors that cause large additional expenses to software producers during the maintenance [1], [2]. The costs related to testing are on the rise; the software industry has identified a need for reducing the growing cost of test environment management [3].

The objective of our study was to explore the software organizations' testing practices, tools and development process models to give an up-to-date picture of industry practices. In addition to answering these questions, this study is also a continuation study to our previous surveys (year 2009 [4] and year 2005 [5]) on the testing practices and test automation in the software industry. Comparison between earlier and current observations reveal changing practices.

The actual testing practices of the software industry were observed via an online survey, conducted in the beginning of 2017. We surveyed organizational units (OU) representing different sizes and business domains in software development. The survey questionnaire consisted of multi-choice, multi-item questions to collect quantitative data for statistical analysis and of open-ended questions for qualitative analysis. This mixed methods study [6] facilitated triangulation of the results [7]. Both the collected quantitative and qualitative data were used to assess the current practices, and compare our new results against our earlier survey results conducted seven years

ago. According to the results, the applied software development models seemed to have shifted towards agile practices, causing changes in the testing infrastructure and test phases' emphasis. The number of automated tools in testing was rising, while the use of the formal process models and capability-maturity models were generally declining.

The work is structured as follows: In Section 2, related surveys and studies are introduced. Section 3 discusses the applied research method used in this work, and Section 4 presents the actual survey results with comparison to the results of our earlier survey. Discussion and conclusions are given in Sections 5 and 6, respectively.

II. RELATED WORK

In addition to our earlier industry-wide survey of test automation and testing practices [4], software testing and test process improvement have been studied by others, for example, Ng et al. [8] in Australia and Chen et al. [9] in China. The study by Ng et al. applied the survey method to establish knowledge on such topics as testing methodologies, tools, metrics, standards, and training. Their study indicated that the most common barrier to developing testing was the lack of expertise in adopting new testing methods and the costs associated with testing tools; also in their study, only 11 organizations reported that they met the testing budget estimates. In a similar vein, Torkar and Mankefors [10] surveyed different types of communities and organizations. They found that 60% of the developers claimed that verification and validation were the first to be downgraded in cases of serious resource shortages during a project.

As for the industry studies, a similar study approach has previously been used in other areas of software engineering. For example, Ferreira and Cohen [11] completed a technically similar study in South Africa, although their study focused on the application of agile development and stakeholder satisfaction. Similarly, Li et al. [12] conducted research on the Commercial Off-The-Self (COTS) based software development process in Norway, Chen et al. [9] studied the application of open source components in software development in China, and Belt et al. [13] surveyed major Scandinavian telecom companies to identify the challenges of testing. Overall, case studies covering entire industry sectors are not particularly uncommon [14], [15].

On longitudinal studies in the development of testing practices, Garousi and Varma [16] conducted a series of surveys in the province of Alberta in Canada. They observed that from 2004 to 2009, the industry transitioned with a distinct elevation of codified practices: all V-model [16] levels of testing work (unit, system, and acceptance) increased along with the level of applied test automation. In addition, the amount of systematic training for the test personnel increased in all of the measured categories. Garousi and Zhi continued the work in 2013 with a nation-wide follow-up survey on the actual software testing practices, where they observed that new tools and development practices have been adopted in the Canadian industry since the prior study [17].

A study of testing practices by Lee, Kang and Lee [18] surveyed the amount of applied testing tools and test practices in South Korea. Their study reveals that even within the last ten years, some software organizations (12% of answers) have not had any meaningful test process or applied any test methods in practice. Interestingly, Lee, Kang and Lee also observed that in their survey population, application of system testing practices was more common than unit testing. One offered explanation was that unit testing is low level activity conducted by the developers, so it does not require separate tools or a process to be followed.

Khosla [19] estimated that in the near future, 80% of the staff in IT departments may be replaced by “artificial intelligence (AI) type systems.” This estimate highlights, for example, automatic collection of run-time data, AI analysis of collected data together with testing and deployment automation during maintenance. Gartner report [20] also emphasizes the importance of automation. According to the Gartner report, software development phases cover 8 % and the maintenance phase, consisting of, for example, defect fixing, testing and deployment of new versions, covers 92 % of the total life cycle costs.

III. RESEARCH METHOD

The survey method described by Fink and Kosecoff [21] was used as the research method in this study in both of the surveys, in 2009 and again in 2017. The objective for a survey is to collect information from people about their feelings and beliefs. Further, surveys are most appropriate when information should come directly from the people [21]. Kitchenham et al. [22] divide comparable survey studies into exploratory studies from which explanations and estimates can be drawn, and confirmatory studies from which strong conclusions can be drawn. We consider this study as an exploratory, observational, and cross-sectional study that explores software testing practices and software quality approaches applied in the software industry.

The 2017 online survey questionnaire included eleven chapters containing questions of organization profile, software testing, test process maturity, applied process models and the tasks related to software development. The constructs were divided into multi-item questions based on, for example, theory, definitions or best practices of the construct. Multi-item questions are questions that are constructed by several items that measure one underlying

construct. Chapters in the questionnaire were planned so that combining respondent’s answers yielded holistic information of the surveyed organizational unit.

To facilitate comparison between our earlier and current survey, seven of the questionnaire chapters were taken directly from our earlier survey [4] which also observed testing and quality assurance practices. The design of the original data collection questionnaire was done by seven researchers from two different research groups. Two additional people were involved in the testing of the questionnaire with test interviews. The questionnaire for the data collection in 2017 was compiled by three researchers, and tested with representatives of our partner organization. The survey questionnaires from both 2009 and 2017 are available in the online appendix at <https://doi.org/10.5281/zenodo.803995>.

The 2009 and the 2017 survey both use the five point Likert scale: 1 fully disagree – 3 neutral – 5 fully agree. The 2017 survey was launched as a web survey via Webropol [23]. The sampling method was probability sampling. The survey was advertised in social media platforms such as LinkedIn, Facebook, Twitter and Researchgate, and by direct contacts to our industrial partners and open calls for participation in several public online discussion channels.

The survey results were analyzed with the R statistical language and its statistics (“stats”) library [24]. In the statistical analysis, survey responses were also treated as single-item and not full constructs to see if the distribution of data between 2009 and 2017 had changed with any statistical significance. Descriptive statistics, displayed in more detail in the online appendix, were generated with the psych R library [25]. When analyzing interval data with the Mann-Whitney U statistical test, continuity correction was enabled to compensate for non-continuous data [26].

To estimate the sample size for our survey we used publicly available statistics provided by the Ministry of Economic Affairs and Employment of Finland. According to the latest report of the software business sector from 2014 [27] there were 3360 companies whose main line of business was software production. The survey questionnaire was opened 930 times and it collected 33 unique responses from respondents working in different organizations within the four-week period it was available in January 2017. This gives the survey a response rate of 3.5 percent, which is fairly normal for Internet surveys according to Fink [21]. In comparison, the 2009 survey had 31 respondents from different software development organizations. This also indicates that both of the surveys had similar-sized sample of the software industry which also, while acknowledging some limitations similar to Iivari [15], were sufficient samples of the industry, and could be analyzed with quantitative approaches.

The survey was anonymous. To identify clusters and to classify answers we collected general information of the organizational unit. This information helped us to classify qualitative answers of the open-ended questions to quantitatively observed clusters. The objective of the study was not to collect data from a certain country but to reveal possible changes in the industry practices.

IV. SURVEY RESULTS

The survey questionnaire included general information of the organizational unit, a number of multi-choice, multi-item questions and open-ended questions. The multi-item questions was estimated by using the Cronbach alpha in the earlier surveys: The Cronbach coefficient alpha expresses the degree to which items in a scale are homogeneous.

Questions concerned the development practices and the available quality assurance infrastructure. In this section we present the survey results collected in January 2017. The results were compared against the 2009 results. We use mode as the primary indicator for individual items in the questionnaire, as the survey questions used an interval Likert scale. Additionally, we performed statistical analysis for the items of the multi-choice questions. In the following we only present results from the statistical analysis that were significant enough. Our anonymized survey dataset, along with the full statistical analysis, is also published in the online appendix.

General information of the organizational unit revealed that the division between the 33 organizations that took part in the survey was very even; very small, small and medium-sized organizations represented each about 21 percent of the participants, while 36.4 percent were large or very large (more than 250 employee) organizations. Approximately eighty percent of the organizations were private companies, while rest of the participants were government agencies or nonprofit organizations. Organizations focusing mainly on national operations formed 21.2 % of the respondents while 39.3 % of organizations focused mostly on international business. Out of all organizations, 30.3 % of them were in-between national and international scale. Out of all organizations, 18.2 % also considered themselves solely or primarily as open source developers. Of the people who responded to our survey, a majority (66.7 %) considered themselves primarily as software developers, while 12.1% had a management position and 15.2% worked in quality assurance. As for the mission-criticality of the organizations, 51.5 % of the organizations reported that product fault could cause remarkable economic losses. Two of these organizations indicated that a fault in their product could cause a loss of a human life. The profiles of the respondent's OUs are shown in Table 1.

The use of testing tools was measured by the question, application level of different software testing tools, and changes were observed through comparison to the earlier results. In this survey, a tool was defined as "an application, framework, web service, extra library, feature of your development environment etc. whichever supports completing the mentioned task".

Table 2 presents the number of used tools is illustrated as percentages in 2017 and 2009. As observable, the three most popular tool categories include defect reporting tools, test automation tools and unit testing tools. Defect/code tracing tools are used by over half of all surveyed organizations. When comparing the new data with the 2009 data, the overall popularity of testing tools has increased in most categories, in particular, test automation, tracing tools and defect reporting. Since 2009,

TABLE I. THE PROFILE OF THE 2017 SURVEY RESPONDENTS (N = 33)

Category	% of respondents
Very Small organization (1-10 employees)	21.2 %
Small (11-50 emp.)	21.2 %
Medium (51-250 emp.)	21.2 %
Large or very large (250+ emp.)	36.4 %
Private company	78.8 %
Government or non-profit organization	21.2 %
Open source developer organization	18.2 %
Primarily national business/operations	30.3 %
Primarily service business	45.5 %
Primarily product business	39.4 %
Mission-critical organization (remarkable economic losses or loss of human life)	51.5 %

TABLE II. THE PERCENTAGE OF APPLIED TESTING TOOLS IN THE INDUSTRY

Tool	% of respondents	
	2017	2009
Bug/Defect reporting	72.7 %	22.6 %
Test automation	66.7 %	29.0 %
Unit testing	57.6 %	38.7 %
Bug/Code tracing	57.6 %	3.2 %
Performance testing	48.5 %	25.8 %
Test case management	45.5 %	48.4 %
Integration testing	45.5 %	16.1 %
Virtual test environment	42.4 %	12.9 %
Quality control	36.4 %	19.4 %
Automated metrics collector	36.4 %	3.2 %
System testing	27.3 %	9.7 %
Security testing	24.2 %	3.2 %
Test completeness	24.2 %	6.5 %
Test design	15.2 %	22.6 %
Protocol/Interface conformance tool	9.1 %	6.5 %

the popularity of test case management (for example, ticketing systems would also fall into this category) remains high, but is no longer the most common testing-specific tool.

The second chapter of the questionnaire discussed the observed test and quality assurance process problems, identified originally in 2009 [4] supplemented with new questions related to maintenance issues. New maintenance and support questions were added because maintenance and support activities have continued growing and are responsible for a large amount of the total lifecycle costs [20]. The observations, especially when comparing the 2009 data with 2017, implied that the configurability of the testing tools has become an issue, and that the support for different software platforms might become an issue, when observing the trend of the changes. Additionally, feature development during late development phases shorten testing schedule and it has become an increasingly pressing issue. The detailed results containing the self-assessment figures for both 2017 and 2009 are presented in Table 3.

The third chapter of the survey was software processes and the amount of agile practices in the organizations. In the survey of 2009, the industry was observed to be interested in the introduction of agile and, in general, more informal practices. Based on our responses, the results of

TABLE III. SOFTWARE TEST PROCESS PROBLEMS, AS IDENTIFIED IN OUR 2017 SURVEY AND IN 2009 [9]. RESPONSES ARE ON A SCALE OF 1 TO 5 (1 FULLY DISAGREE - 3 NEUTRAL - 5 FULLY AGREE)

	2017 mode	2009 mode
Complicated testing tools cause test configuration errors.	4	1
Commercial testing tools do not offer enough support for our development platforms.	3	1
It is difficult to automate testing because of low reuse and high price.	4	5
Insufficient communication slows the bug-fixing and causes misunderstanding between testers and developers.	4	2
Feature development in the late phases of the product development shortens testing schedule.	4	4
Testing personnel do not have expertise in certain testing applications.	4	4
Existing testing environments restrict testing.	3	4

this chapter are very in-line with the earlier results giving emphasis on the agility of the industry-applied processes.

The industry drive towards agile practices can also be observed from another chapter in our survey where we asked about the use of formal process models such as SPICE (software process assessment, ISO/IEC 15504, currently part of the ISO/IEC 33000 series) [28] or software testing standard (ISO/IEC 29119) [29]. The question covered also the utilization of capability and maturity models, such as TMMi - test maturity model integrated [30] or CMMi – capability maturity model integrated [31]. Based on our survey results, the use of formal models have decreased within the last eight years. Some form of process model (formal or self-defined) was applied by 21.2 percent of organizations (62.5 percent in 2009), while none of the organizations in 2017 applied capability or maturity certificates in their organization (it was 43.8 percent in 2009). In 2017, V-model, acceptance criteria for tickets and “generic agile” were mentioned, all based on best practices collected from various sources and “self-defined”. Detailed division of answers is presented in Table 4.

The final chapter in the survey included questions concerning the software testing and quality assurance

practices. In general, the results do not indicate any major shifts in the applied testing and quality assurance practices between the two surveys. Organizational units are confident that they are building the product right, and at the same time, building the right product. Survey responses detailed in Table 5 highlights some differences between the surveys: Testing schedules may not be kept (2009 mode 4, partially agree, 2017 mode 2, partially disagree) and time is not necessarily allocated enough for testing (2009 mode 4, partially agree, 2017 mode 2, partially disagree). Respondents are less confident in their function testing practices (3.8 vs. 2.9 in average between 2009 and 2017. 2009 mode 4, partially agree, 2017 mode 3, neutral). Statistical significance in the difference of distributions between the years for the single question “our functional testing is excellent” can be established with the Mann–Whitney U test, $U=613$ at significance level $p=0.005$. Formal inspections are the testing practices on which the surveyed organizations have become more confident (2009 mode 2, partially disagree, 2017 mode 4, partially agree), while code review practices have become more varied between different organizations (2009 mode 4, partially agree, 2017 mode 1, fully disagree).

In addition to multi-choice questions the survey contained open-ended questions, where we asked the respondents to explain how their organization manages the increasing testing and maintenance effort. The following themes were highlighted from the responses:

- Moving from proprietary software to open source
- Increasing the coverage of automated tests
- Focusing on service scalability in design
- Re-implementing legacy applications
- Setting up dedicated testing and development environments
- Offshoring testing work
- Establishing pre-planned maintenance time for projects, during which last defects are fixed
- Forming dedicated maintenance teams
- Emphasizing the responsibility of current developers
- Employing a risk-based testing approach to cover the most critical components rather than trying to get perfect coverage.

TABLE IV. THE USE OF FORMAL PROCESS MODELS AND CAPABILITY OR MATURITY CERTIFICATES IN ORGANIZATIONS

Category	2017	2009
Process model - Yes, formal	9.1 %	25.0 %
Process model - Yes, informal	12.1 %	37.5 %
Process model - No	63.6 %	37.5 %
Capability certificate - Yes, formal	37.0 %	0.0 %
Capability certificate - Yes, informal	6.3 %	0.0 %
Capability certificate - No	56.3 %	81.8 %

TABLE V. THE SELF-ASSESSMENT OF THE QUALITY OF THE DIFFERENT TESTING AND QUALITY ASSURANCE PRACTICES (1 FULLY DISAGREE – 3 NEUTRAL – 5 FULLY AGREE)

	2017 mode	2009 mode
Our software correctly implements a specific function. We are building the product right.	4	5
Our software is built traceable to customer requirements. We are building the right product.	5	4
Our formal inspections are OK.	4	2
We go through checklists.	2	3
We keep code reviews.	1	4
Our unit testing (modules or procedures) is excellent.	4	2
Our integration testing (multiple components together) is excellent.	3	3
Our usability testing (adapt software to users' work styles) is excellent.	3	2
Our function testing (detect discrepancies between a program's functional specification and its actual behavior) is excellent.	3	4
Our system testing (system does not meet requirements specification) is excellent.	3	4
Our acceptance testing (users run the system in production) is excellent.	4	4
We keep our testing schedules.	2	4
Last testing phases are kept regardless of the project deadline.	4	4
We allocate enough testing time.	2	4

V. DISCUSSION AND IMPLICATIONS

The objective of this study was to explore the testing practices of software companies, compare the results with earlier survey result from the year 2009 and thereby outline changes in software industry. The collected data is publicly available in the online appendix should other researchers want to validate, replicate or build upon our findings.

Overall, the availability and application level of testing- and quality assurance-dedicated tools has increased across the industry, in almost all measured categories. Especially tools related to automated testing (e.g. test automation, automated metrics collection, performance testing, tracing tools) have increased significantly. The respondents of the survey refer to testing and automated testing almost synonymously. The available testing tools in 2017 are more sophisticated than in the 2009, imposing less restrictions but causing more configuration problems.

The use of different formal standards, certifications and process models has decreased, while the amount of agile practices has increased moderately. The mission-criticality of the software no longer limits the organization from using agile practices or other informal approaches. In 2017, the last product features are introduced later during the development process than in 2009. This leads to increased shortages of testing resources (time) and puts more emphasis on the acceptance phase testing. Test design and documentation work in general have declined while the confidence in functional testing practices has declined. Issues in testing and maintenance are more related to software development processes and practices, the quality and coverage of testing, and test schedule rather than the cost of quality assurance.

The survey results indicate increase in test automation, a shift towards agile practices, and that the formal software process models are less popular among industry practitioners. Results are in line with the observations of, for example, Khosla [19]: the rise of automation in testing, deployment and maintenance. Growing test automation also fits well to the observations of the Gartner report [20]. Explanatory factors to the growing test automation include, for example, agile methods with regression testing [32], continuous deployment and integration to

shorten the timespan between product versions [33], DevOps to lower the threshold between development and use [34], and the general requirements for automation in IT-departments, server rooms and data centers to reduce the costs [19].

In comparison to other industry surveys in software testing, our results suggest similar trends as, for example, Canadian software industry report by Garousi and Zhi [17]. The most important testing tools in our study include defect tracking, unit testing and test automation, and Canadian organizations see functional and unit testing as the most common testing work. Likewise, Canadian organizations perform testing activities mostly during a dedicated testing phase in development (test-last approach). Our respondents did not suggest any other approach than test-last, and our results indicate that test phases may even be skipped in some circumstances.

Formal process models are more common in large and very large organizations. According to the study of Hardgrave and Armstrong [35], small and medium-sized organizations are able to apply the principles and best practices of the formal models in their work. Therefore, the reason for the decreasing use of the process models cannot be directly explained, and has to be assessed in more detail in the future works.

Concerning the validity of the study, even though the survey constructs and questions between the rounds were almost the same, there were differences in the data collection procedures: in 2009 the data was collected by interviewing representatives of software organizations whereas the 2017 dataset was collected online. The number of interviews in the 2009 dataset was 32 and the number of filled on-line questionnaires in the 2017 survey was 33. The response rate of 2017 was in line with the estimates given for on-line surveys [21]. The sample is small but comparable with the sample of 2009 and the observations are presented as explorative and not as strong conclusions. Overall, the metrics presented in this paper are accumulation data from the survey, so the researcher bias on the results should be minimal. The 2017 results were largely similar to the 2009 results, which adds to the rigor of the results, and helps highlight differences between the years.

VI. CONCLUSION

The results of the survey presented in this paper indicate that the software testing practices have undergone some changes in the industry within the last eight years. First, automation in testing has continued its growth. Within testing trends, automation has become more common on all levels of testing. Second, the application of formal software process models and capability maturity models seems to have decreased, while the testing tools have become increasingly common and more sophisticated.

This change is also reflected by the organizational considerations over the testing tools: the tools no longer restrict the organizational unit as much as they did in 2009 but in exchange, configuration problems and lack of platform support have become increasingly common. Also testing done during the design phase is decreasing. Since the last features are introduced later in the software development process, the emphasis on the late testing and, especially, acceptance testing has increased, while, at the same time, available time for testing work has decreased. Overall, the changes are not dramatic but the industry practices evolve as we can observe from the comparison of the surveys.

In our future work, the focus is on the expenses of testing and quality assurance. Based on our observations, the reduced use of formal processes, and the need to push new features into the product, mean that the products need better support for acceptance testing, regression testing and in general quality assurance for the features added after the initial launch. This study area is interesting, since the reduction of the costs of the maintenance cycle and automated regression testing would probably have a meaningful impact on the overall costs of quality assurance work.

REFERENCES

- [1] E. Kit, *Software testing in the real world*. Addison-wesley, 1995.
- [2] S. Planning, "The economic impacts of inadequate infrastructure for software testing," 2002.
- [3] "World Quality Report 2016-17," Capgemini Group, 2017.
- [4] J. Kasurinen, O. Taipale, and K. Smolander, "Software test automation in practice: empirical observations," *Adv. Softw. Eng.*, vol. 2010, 2010.
- [5] O. Taipale, K. Smolander, and H. Kälviäinen, "Finding and Ranking Research Directions for Software Testing," in *Software Process Improvement: 12th European Conference, EuroSPI 2005, Budapest, Hungary, November 9-11, 2005. Proceedings*, I. Richardson, P. Abrahamsson, and R. Messnarz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 39–48.
- [6] G. Paré and J. J. Elam, "Using case study research to build theories of IT implementation," in *Information systems and qualitative research*, Springer, 1997, pp. 542–568.
- [7] N. K. Denzin, *The research act: A theoretical introduction to sociological methods*. Transaction publishers, 1973.
- [8] S. P. Ng, T. Murnane, K. Reed, D. Grant, and T. Y. Chen, "A preliminary survey on software testing practices in Australia," in *Software Engineering Conference, 2004. Proceedings. 2004 Australian*, 2004, pp. 116–125.
- [9] W. Chen, J. Li, J. Ma, R. Conradi, J. Ji, and C. Liu, "An empirical study on software development with open source components in the chinese software industry," *Softw. Process Improv. Pract.*, vol. 13, no. 1, pp. 89–100, 2008.
- [10] R. Torkar and S. Mankefors, "A survey on testing and reuse," in *Software: Science, Technology and Engineering, 2003. SwSTE'03. Proceedings. IEEE International Conference on*, 2003, pp. 164–173.
- [11] C. Ferreira and J. Cohen, "Agile systems development and stakeholder satisfaction: a South African empirical study," in *Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology*, 2008, pp. 48–55.
- [12] J. Li, F. O. Bjørnson, R. Conradi, and V. B. Kampenes, "An empirical study of variations in COTS-based software development processes in the Norwegian IT industry," *Empir. Softw. Eng.*, vol. 11, no. 3, pp. 433–461, 2006.
- [13] P. Belt, J. Harkonen, M. Mottonen, P. Kess, and H. Haapasalo, "Improving the efficiency of verification and validation," *Int. J. Serv. Stand.*, vol. 4, no. 2, pp. 150–166, 2008.
- [14] K. Y. Wong, "An exploratory study on knowledge management adoption in the Malaysian industry," *Int. J. Bus. Inf. Syst.*, vol. 3, no. 3, pp. 272–283, 2008.
- [15] J. Iivari, "Why are CASE tools not used?," *Commun. ACM*, vol. 39, no. 10, pp. 94–103, 1996.
- [16] V. Garousi and T. Varma, "A replicated survey of software testing practices in the Canadian province of Alberta: What has changed from 2004 to 2009?," *J. Syst. Softw.*, vol. 83, no. 11, pp. 2251–2262, 2010.
- [17] V. Garousi and J. Zhi, "A survey of software testing practices in Canada," *J. Syst. Softw.*, vol. 86, no. 5, pp. 1354–1376, May 2013.
- [18] J. Lee, S. Kang, and D. Lee, "Survey on software testing practices," *IET Softw.*, vol. 6, no. 3, pp. 275–282, 2012.
- [19] V. Khosla, "Keynote speech," presented at the Structure Conference, San Francisco, 2016.
- [20] "The Four Laws of Application, Total Cost of Ownership." Gartner, 2012.
- [21] A. Fink, *How to Conduct Surveys: A Step-by-Step Guide*. Sage Publications, 2012.
- [22] B. A. Kitchenham *et al.*, "Preliminary guidelines for empirical research in software engineering," *IEEE Trans. Softw. Eng.*, vol. 28, no. 8, pp. 721–734, 2002.
- [23] "Webropol." [Online]. Available: <https://www.webropol-surveys.com/>. [Accessed: 22-Jun-2017].
- [24] R Core Team, *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing, 2017.
- [25] W. Revelle, *psych: Procedures for Psychological, Psychometric, and Personality Research*. Evanston, Illinois: Northwestern University, 2017.
- [26] R. Bergmann, J. Ludbrook, and W. P. Spooren, "Different outcomes of the Wilcoxon—Mann—Whitney test from different statistics packages," *Am. Stat.*, vol. 54, no. 1, pp. 72–77, 2000.
- [27] T. Metsä-Tokila, "Ohjelmistoala," Työ- ja elinkeinoministeriö, 2014.
- [28] "ISO/IEC 15504-1: Information technology — Process assessment — Part 1: Concepts and vocabulary." International Organization for Standardization, 2004.
- [29] "ISO/IEC 29119-2: Test Processes." International Organization for Standardization, 2013.
- [30] E. van Veenendaal and B. Wells, *Test Maturity Model Integration TMMi*. The Netherlands: Uitgeverij Tutein Nolthenius, 2012.
- [31] R. Kneuper, *CMMI: Improving Software and Systems Development Processes Using Capability Maturity Model Integration*. Rocky Nook, 2008.
- [32] K. Schwaber and M. Beedle, *Agile software development with Scrum*, vol. 1. Prentice Hall Upper Saddle River, 2002.
- [33] M. Fowler, "Continuous Integration," martinfowler.com. [Online]. Available: <https://martinfowler.com/articles/continuousIntegration.html>. [Accessed: 22-Jun-2017].
- [34] S. K. Bang, S. Chung, Y. Choh, and M. Dupuis, "A grounded theory analysis of modern web applications: knowledge, skills, and abilities for DevOps," in *Proceedings of the 2nd annual conference on Research in information technology*, 2013, pp. 61–62.
- [35] B. C. Hardgrave and D. J. Armstrong, "Software process improvement: it's a journey, not a destination," *Commun. ACM*, vol. 48, no. 11, pp. 93–96, 2005.

Publication II

Hynninen, T., Kasurinen, J., Knutas, A., and Taipale, O.

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

Journal of Software Engineering and Applications.

Vol. 11, no. 4, pp. 139–152, 2018

This open-access article is distributed under the terms of the Creative Commons Attribution 4.0 International License (CC-BY 4.0)

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

Timo Hynninen¹, Jussi Kasurinen², Ossi Taipale¹

¹School of Business and Management, Lappeenranta University of Technology, Lappeenranta, Finland

²South-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

Copyright © 2018 by authors and Scientific Research Publishing Inc. This work is licensed under the Creative Commons Attribution International License (CC BY 4.0).

<http://creativecommons.org/licenses/by/4.0/>



Open Access

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 & Zaroni [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**.

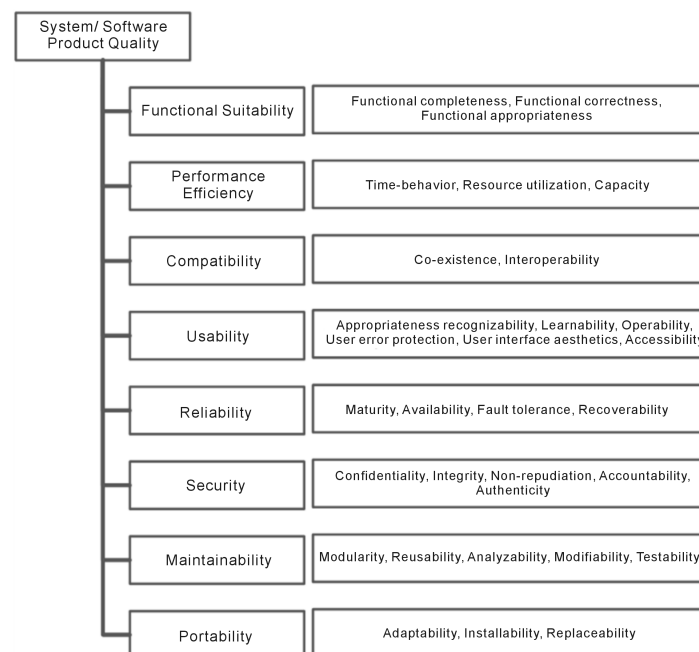


Figure 1. The ISO/IEC 25010 software product quality model [7], characteristics on left, and the subcharacteristics on the right.

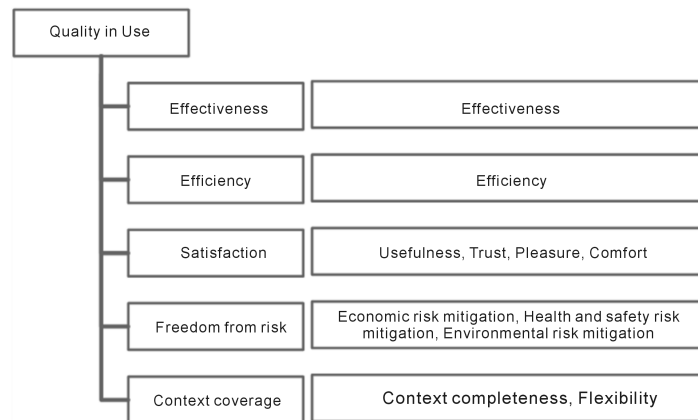


Figure 2. ISO/IEC 25010 software quality-in-use model [7].

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 criteria 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

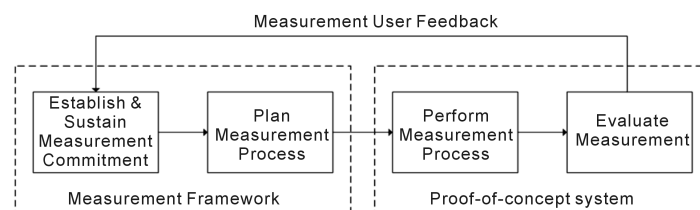


Figure 3. The measurement framework and proof-of-concept system (Modified from ISO/IEC 15939 Systems and software engineering—Measurement process model [24]).

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, Modifiability)	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

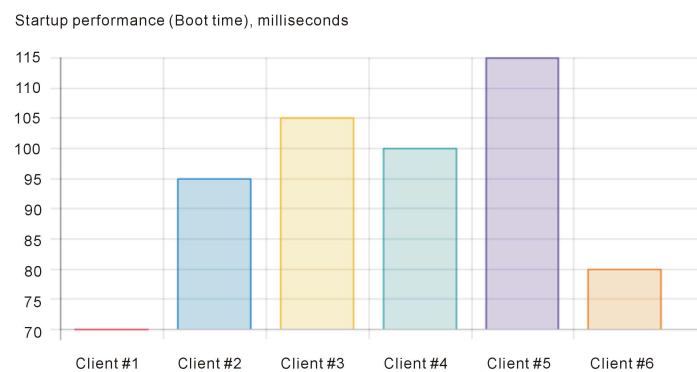


Figure 4. Example, a time-performance metric collected from six different clients in a test scenario.

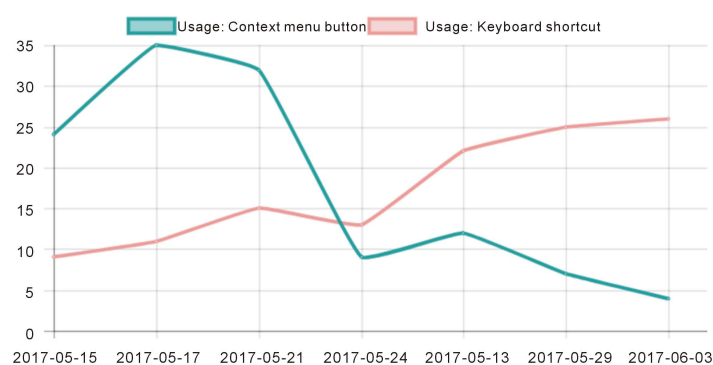


Figure 5. A feature utilization metric collected from clients in a test scenario.

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 modeling 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 *29th 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/ijismd.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 37th 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 4th 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. *16th 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. *22nd 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. *10th 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., Boston.
- [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>

Publication III

Savchenko, D., Hynninen, T., and Taipale, O.
Code quality measurement: case study

In Proceedings of the 2018 41st International Convention on Information and
Communication Technology, Electronics and Microelectronics (MIPRO)
Pp. 1455–1459, 2018
© 2018, IEEE

Code quality measurement: case study

D. Savchenko*, T. Hynninen* and O. Taipale*

* Lappeenranta University of Technology, Lappeenranta, Finland
dmitrii.savchenko@lut.fi, timo.hynninen@lut.fi, ossi.taipale@lut.fi

Abstract - As it stands, the maintenance phase in the software lifecycle is one of the biggest overall expenses. Analyzing the source code characteristics and identifying high-maintenance modules is therefore necessary. In this paper, we design the architecture for a maintenance metrics collection and analysis system. As a result, we present a tool for analyzing and visualizing the maintainability of a software project.

Keywords - maintenance, code quality

I. INTRODUCTION

Maintenance and upkeep is a costly phase of software life cycle. It has been estimated that maintenance can reach up to 92% of total software cost [1]. Code quality can be analyzed using various existing metrics, which can give an estimate on the maintainability of software. There are several tools and frameworks for assessing the maintainability characteristics of a project. Many tools are included in integrated development environments (IDEs), such as Eclipse metrics [2], JHawk [3] or NDepend [4]. As such the existing tools are specific to platform and programming language, providing quality analysis during development. Considering maintenance also includes activities post-release of a software product, it would be beneficial to perform quality measurement also in the maintenance and upkeep phase of life cycle.

One solution to the post-release monitoring are online data gathering probes, which can be inserted into production code to gather runtime performance data. In order to establish and sustain a commitment for maintenance measurement this work introduces a design for data collection and storage. In this paper we present an architecture for systematically collecting code metrics for maintenance. Additionally, the visualization and analysis of the metrics are explored.

In this study we will focus on the analysis of web-applications. This delimitation is due to the collection of runtime metrics as well as static metrics. The focus on web-applications provides a reasonably standardized measurement interface for runtime performance through the browser's web API. In this paper we also propose the design and implementation of the system called Maintain. The probes for gathering metrics in the system are implemented in both JavaScript and Ruby programming languages.

Rest of the paper is structured as follows. In Section 2, related work in analyzing software maintainability is

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

introduced. Sections 3 and 4 presents the architecture and our implementation for a metrics collection and analysis system, which is main contribution of this work. Evaluation of the system's performance and utility is presented in Section 5. Finally, discussion and conclusions are given in Section 4.

II. RELATED RESEARCH

Software maintenance, as defined by ISO 14764 standard, is the "the totality of activities required to provide cost-effective support to a software system", consisting of activities both during development and post-release [5]. The analysis of software maintainability is by no means a novel concept. Motogna et al. [6] presented an approach for assessing the change in maintainability. In [6], metrics were developed based on the maintainability characteristics in the ISO 25010 software quality model [7]. The study presents how different object oriented metrics affect the quality characteristics.

A study by Kozlov et al. [8] distinguished that particular code metrics (data variables declared, McClure Decisional Complexity) have strong correlations with the maintainability of a project. In the work, the authors analysed the correlation between maintainability and the quality attributes of a Java-project.

In the study by Heitlager et al. [9] a practical model for maintainability is discussed. The study discusses the problems of measuring maintainability, particularly with expressing maintainability as a single metric (Maintainability index).

Studies where different evaluation methods are combined in order to get a more thorough view on the maintainability of a project have been conducted during the past decade. For example, Yamashita [10] combined benchmark-based measures, software visualization and expert assessment. In a similar vein, Anda [11] assessed the maintainability of a software system using structural measures and expert assessment. In general, these studies suggest that visualization systems providing developers and project managers with an analysis of the health of a software project can help distinguish problematic program components, and thus help in the maintenance efforts of software.

III. ARCHITECTURE

Maintain system architecture is presented at the figure 1. System consists of the following components:

- *Probe* is a program that gathers some valuable data from the software (static or dynamic). Each probe should have an associated analyzer;
- *Data Storage* – data storage that stores the raw data from the probes. It also has REST interface that receives the data from the probes;
- *Analyzer* is a program that gets the raw data from the associated probe and creates a report, based on this data;
- *Report Storage* – data storage that stores reports from analyzers;
- *Report Visualizer* is a component that creates a visual representation of the report.

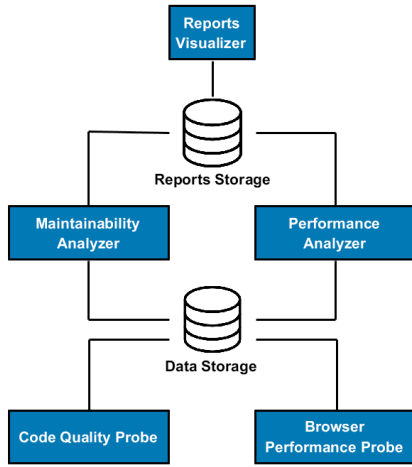


Figure 1. System architecture

Workflow of the system is centered around the Data Storage. Generally, it looks like this:

- Probes gather the information from the source code, it might be some static analysis results or dynamic performance data;
- Gathered and normalized data is sent to the Data Storage. Probe can have different data types, data structure is defined by analyzer;
- When new data is received by Data Storage, the associated analyzer is called. It requests the data from the Data Storage, produces report (object, that contains current status of the analyzed application aspect and a set of time series for the end user);

When end user requests the report, Reports Visualizer generates a visual representation of the time series, that were created by analyzers.

IV. IMPLEMENTATION

Maintain system was implemented using Ruby on Rails framework and hosted on Heroku cloud platform. Project details page is shown on Figure 3. This page provides the information about the current state of the

project, that is described as a set of 8 scores, based on quality characteristics, described in ISO/IEC 25010 [7]. Those scores are visualized as a polar chart with 8 axis for each quality characteristic respectively. Score calculation is based on the report statuses – each report has an associated probe, and each probe has a set of associated quality characteristics. Quality characteristics are set by the project administrator.

System class structure is organized as pictured in figure 2. As system gathers the data using REST API, it is generally impossible to predefine all possible probes and probe types and set their quality characteristics in advance. That's why we decided to let user define the quality characteristics for probe when it is created or modified. Result score is based on statuses of last reports for each probe respectively.

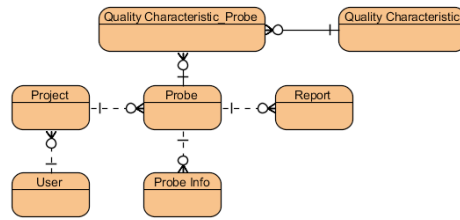


Figure 2. System entity-relationship diagram

A. Probes

As a case study, we have implemented four probes: HAML, JavaScript and Ruby code quality probes, and browser performance probe. JavaScript and Ruby code quality probes are based on maintainability index, which is calculated using the following formula:

$$\text{maintainability} = 171 - (3.42 * \text{Math.log}(\text{effort})) - (0.23 * \text{Math.log}(\text{cyclomatic})) - (16.2 * \text{Math.log}(\text{loc}))$$

HAML maintainability index uses recursive formula, based on linter report:

$$\text{Maintainability} = a * \text{maintainability}$$

where **a** is 0.9 for linter error and 0.99 for linter warning

Code quality probes produce the following data for Data Storage:

```

{
  maintainability: M,
  revision: R,
  datetime: D,
  modules: Ms
}
  
```

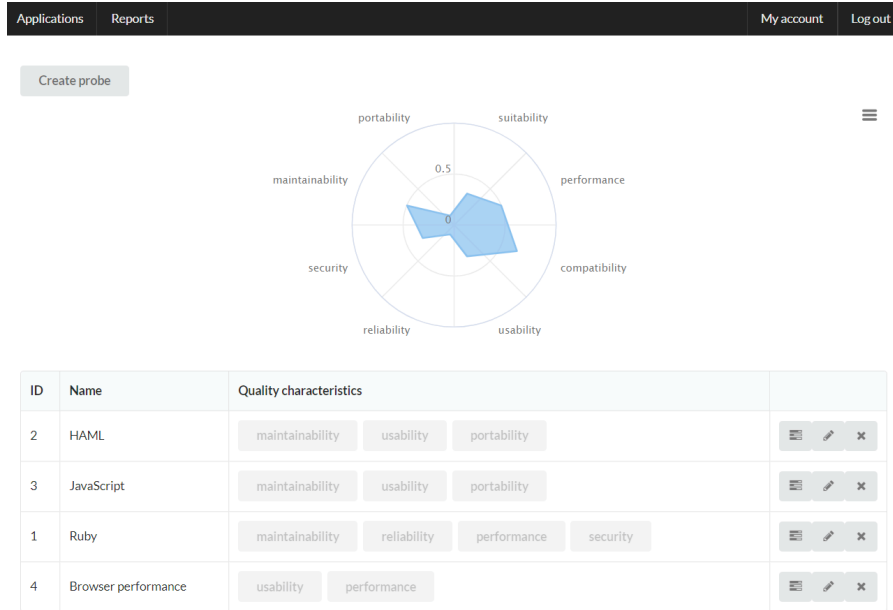


Figure 3. Project page example

where M is average maintainability index for the whole project, R is current Git revision, D is current date and time, and Ms is a list of maintainability index for project files and their names. Browser performance probe generates report in different format:

```
{
  page: P,
  timing: T,
  datetime: D
}
```

Where P is an URL of the current page (without query), T is the time between page load start time and DOM ready event time in milliseconds, and D is current date and time.

B. Analyzers

Currently we have implemented two different analyzers - maintainability analyzers for Ruby, JavaScript and HAML probes, and performance analyzer for browser performance probe. Workflow for maintainability analyzer works is described below:

- Data from Data Storage is grouped by days, maintainability index for each day calculated as a median of indices for day. If no data presented for day, analyzer sets the value for the previous day (fallback for weekends);

- List of maintainability indices are smoothed using exponential moving average method, those values are used as a time series for visualizer;
- Linear regression for last five days is used as a status of the project source code quality: if it is less than zero, then code quality is bad.

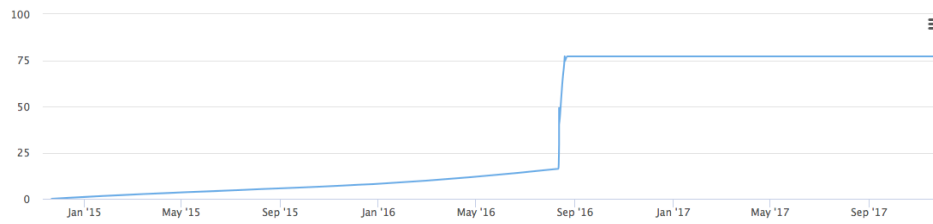
Workflow for browser performance analyzer is different:

- Performance data is grouped by five minutes, value for each section is calculated as a 95th percentile of all values for section;
- If values for all sections are less than 2 seconds, then browser performance is good.

V. MAINTAIN SYSTEM USAGE EXAMPLE

Maintain system was evaluated using a proprietary web application, that was implemented using Ruby on Rails as a backend, and CoffeeScript on top of React.JS as a frontend. This project is on maintenance phase, so we decided to analyze historical data and compare Maintenance system results with the feedback from the project manager, who managed the analyzed project. Application was used by 5 administrators and about 10000 users. Maintenance system was deployed in Heroku cloud, while probes were running on local PC, that had 1.8 GHz 2-core CPU and 4 Gb RAM. We gathered the code quality information for all the previous commits to make picture more consistent.

Report for application Innovation Center Portal (JavaScript) from 29.03.18



Report for application Innovation Center Portal (HAML) from 29.03.18

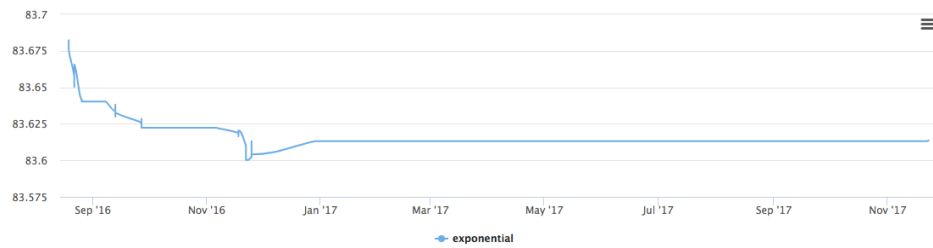


Figure 4. Project code quality measurements

Figure 3 illustrates the general ‘health’ of the analyzed application at the last Git revision at Master branch. Figure 4 shows the JavaScript (CoffeeScript) and HAML code quality. The project was started as a pure backend solution, while frontend development started at the beginning of September 2016. As shown in the graph, HAML code quality was decreasing from September 2016, until December 2016, then it was stable. This behavior can be explained by a deadline of the project, that was at the end of the year 2016. After the deadline, the project active development stopped. Project manager evaluated the results and stated, that such an ‘early warning’ system could notify the team and save some development resources.

VI. DISCUSSION AND CONCLUSION

The objective of this study was to facilitate the systematic collection and analysis of maintenance metrics, in order to reduce the effort required in the maintenance phase of software already during development. To realize the goal we designed and implemented an architecture for a system which can be used to collect both static and runtime metrics of a software project. We then implemented analysis tools to visualize these metrics, and display the most high-maintenance modules in a project repository.

The novelty of the presented work is the extendibility and modularity of the architecture. The architecture is not platform specific. New probes and corresponding analyzers can be added at any stage, using the REST API with any programming language or platform. The data storage and reporting system provide a common interface for the systematic collection of quality metrics, allowing the developers of a project to establish and sustain a commitment for quality measurement.

Providing a platform to establish the measurement commitment is important, because previous research

shows that the quality assurance and testing practices of developers do not necessarily line up with measurement possibilities distinguished in academic research. For example, the recent study by Garousi and Felderer distinguishes that the industry and academia have different focus areas on software testing [12]. Likewise, Antinyan et al. show in [13] that existing code complexity measures are poorly used in industry. In this work, we used the maintainability index as an indicator for code quality, as it has been used in both academia and industry. In future, we should work on evaluating whether quality metrics presented in academic publications could be implemented into our system as probes providing reliable measurements.

Additionally, in future work we aim to develop more measurement probes in the system. We should evaluate the different metrics to distinguish which measurements provide the most useful information about software maintainability.

REFERENCES

- [1] “The Four Laws of Application, Total Cost of Ownership.” Gartner, Inc., 2012.
- [2] Eclipse Metrics Plug-in, <http://sourceforge.net/projects/metrics>. (accessed 5th Feb 2018).
- [3] JHawk, <http://www.virtualmachinery.com/jhawkprod.htm>. (accessed 5th Feb 2018).
- [4] NDepend “<http://www.ndepend.com>”, (accessed 5th Feb 2018).
- [5] ISO/IEC, “ISO/IEC 14764: Software Engineering - Software Life Cycle Processes - Maintenance.” 2006.
- [6] S. Motogna, A. Vescan, C. Serban, and P. Tirban, “An approach to assess maintainability change,” in 2016 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR), 2016, pp. 1–6.
- [7] ISO/IEC, “ISO/IEC 25010: Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models.” 2011.
- [8] D. Kozlov, J. Koskinen, J. Markkula, and M. Sakkinen, “Evaluating the Impact of Adaptive Maintenance Process on Open Source Software Quality,” in First International Symposium on

- Empirical Software Engineering and Measurement (ESEM 2007), 2007, pp. 186–195.
- [9] I. Heitlager, T. Kuipers, and J. Visser, “A practical model for measuring maintainability,” in *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*, 2007, pp. 30–39.
- [10] A. Yamashita, “Experiences from performing software quality evaluations via combining benchmark-based metrics analysis, software visualization, and expert assessment,” in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 421–428.
- [11] B. Anda, “Assessing software system maintainability using structural measures and expert assessments,” in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, 2007, pp. 204–213.
- [12] V. Garousi and M. Felderer, “Worlds Apart: Industrial and Academic Focus Areas in Software Testing,” *IEEE Software*, vol. 34, no. 5, pp. 38–45, 2017.
- [13] V. Antinyan, M. Staron, and A. Sandberg, “Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time,” *Empirical Software Engineering*, pp. 1–31, 2017.

Publication IV

Savchenko, D., Hynninen, T., Taipale, O., Smolander, K., and Kasurinen, J.
Early-warning system for software quality issues using maintenance metrics

International Journal on Information Technologies and Security

Vol. 12, No 4, pp. 35–46, 2020

© 2020, Union of Scientists in Bulgaria

EARLY WARNING SYSTEM FOR SOFTWARE QUALITY ISSUES USING MAINTENANCE METRICS

Dmitrii Savchenko¹, Timo Hynninen², Ossi Taipale¹, Kari Smolander¹ and Jussi Kasurinen¹

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, we analyse how to reduce the cost of software maintenance and manage complexity with an analysis tool that indicates the expected amount of maintenance work based on the first observations after a new release. Based on our initial findings, the maintenance indicators 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], but preventative tasks 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: *is it possible 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 anomaly, 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 developer teams tend to use as much automation as possible, especially in quality assurance (QA), and

use informal software development processes over formal approaches and automation tools over formal 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. A further study by Yamashita [28] points that the systems that combine real-time measurements of the developed software with a visualization can help to develop the software with better quality requiring less maintenance.

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 index 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 [29, 30]. Design science study is usually understood as research that produces constructs, methods, and models, and uses two iterative approaches: building and evaluation [30]. In practice, design science is may be described by the process called design cycle (see Figure 1).

In this study, we initiated the design cycle from the first step, so our study starts from the problem of software maintenance costs and aims at the research question through the development of the prototype framework. We started with the identification of the original problem, that the automated collection of data related to most of the quality-defining attributes of the quality standard ISO/IEC 25010 [18] requires inputs which beyond the reach of a simple data collection or repository mining tools since no suitable data is usually available. For this observation, in the initial design we created the concept of probes, and 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 to test out the idea and demonstrate the concept to other interested parties. After this initial evaluation step we identified further problems and concepts, for the probes needed to run the quality assessment tests with real software development projects.

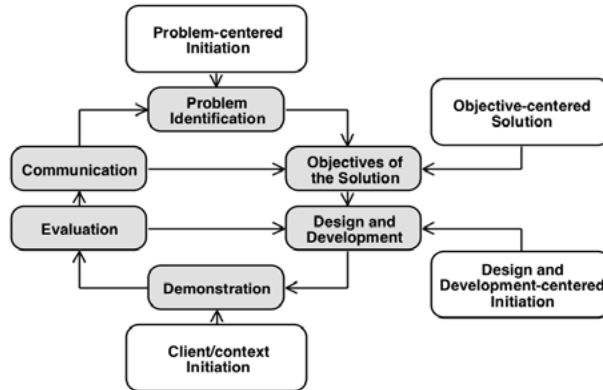


Fig. 1. Design science research method adapted from Peffers et al. [30]

The objective of the Maintain data collection and early warning system is was to reduce the overall software maintenance effort and costs. For this objective, study by Heitlager et al. [27] indicated that the Maintainability index of the source code is linked with software quality, and, therefore, assessing the changes in the maintainability index could affect the overall maintenance costs. 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 [31].

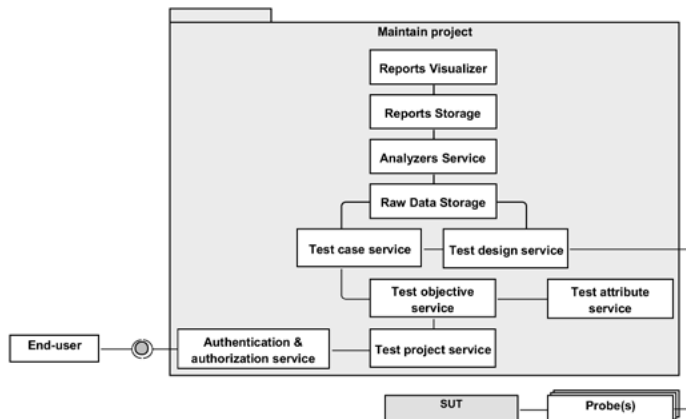


Fig. 2. Maintain-tool architecture

Design science implies both building and evaluation of the developed artefact. To evaluate the .Maintain tool, we ended up implementing the same set of code quality probes for the different programming languages. Finally to test our proof-of-concept tool, we analysed three finished real-life software projects at different

stages of development pulled from the code repository, 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.

4. RESULTS

In our proof-of-concept cases, the objective was to dynamically measure changes in the Maintainability index, and compare this information against the other data sources in the development project. In this stage, we collected and analysed historical data from three different project repositories, and then interviewed the developer teams about their activities to see if our maintainability tools would match the perceived quality changes in the developed systems.

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 [32] found in the analysed file:

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

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:

$$trend = code_quality - linreg(code_quality) \quad (2)$$

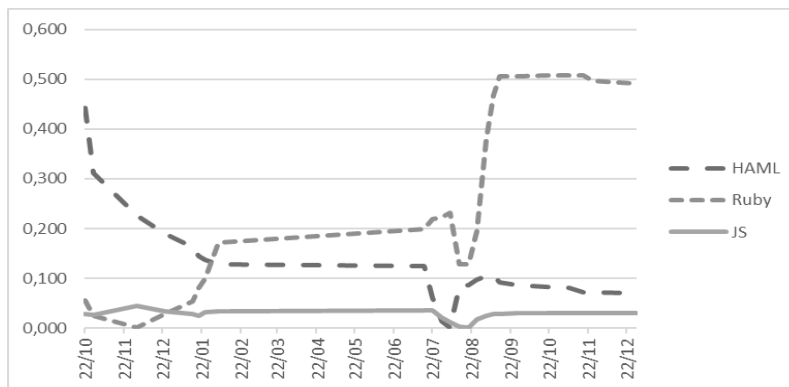


Fig. 3. Project 1 quality report

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.

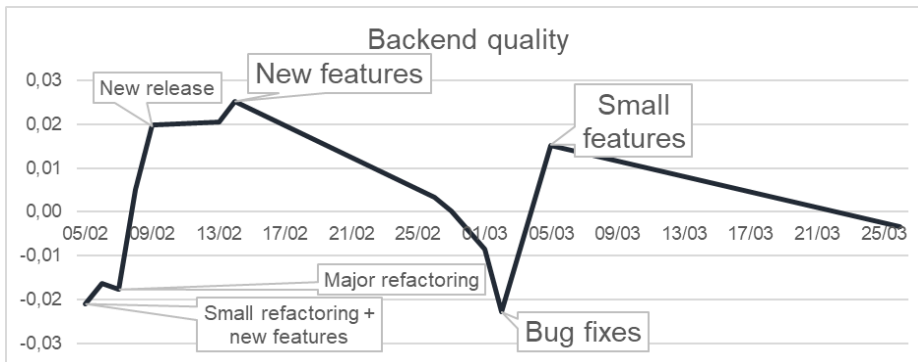


Fig. 4. Project 2 quality report with the comments from project manager report

4.3. Project 3

In the third pilot project with our maintainability observation tools, we decided to analyse 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.

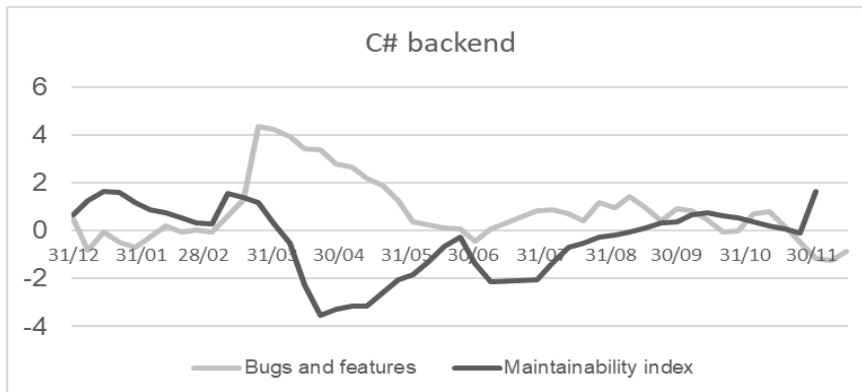


Fig. 5. Project 3 quality report with bugs and features -metric

5. DISCUSSION AND IMPLICATIONS

The modern development processes for software systems emphasize concepts such as continuous integration [11], cloud as the delivery model [33], service computing [33], 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 to the project manager.

Beyond our work with the maintainability estimations, the idea of finding the link between the source code metrics and project quality attracts 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. On the other hand, Kannangara and Wijayanake [34] analysed the difference in the software quality before automated refactoring and after that using several source code metrics, but state that those metrics do not show improvements in the software quality after refactoring. However, for example Fontana [16] indicates that code smell can be useful for software quality estimation and can be used as a quality metric.

To guarantee the validity of the study we used two different approaches in the evaluation and in the methodological triangulation. Robson [36] 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 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 artefact, 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 artefact 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, Swanson EB. Software Maintenance Management. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.; 1980.
- [2] Ma, D. (2007, July). The business model of" software-as-a-service". In Ieee international conference on services computing (scc 2007) (pp. 701-702). IEEE.
- [3] Chen, L. (2015). Continuous delivery: Huge benefits, but challenges too. IEEE Software, 32(2), 50-54.
- [4] Kyte A. The Four Laws of Application Total Cost of Ownership. April 2012. <https://www.gartner.com/doc/1972915/laws-application-total-cost-ownership>. Accessed October 15, 2018.
- [5] Capgemini. WORLD QUALITY REPORT 2017-16.2017.
- [6] Leppänen, M., Mäkinen, S., Pagels, M., Eloranta, V. P., Itkonen, J., Mäntylä, M. V., & Männistö, T. (2015). The highways and country roads to continuous deployment. Ieee software, 32(2), 64-72.
- [7] Banker RD, Datar SM, Kemerer CF, Zweig D. Software complexity and maintenance costs. Commun 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 Systems. ; 2011. doi:10.4018/9781599044149.ch014.
- [9] Glass RL, Collard R, Bertolino A, Bach J, Kaner C. Software testing and industry needs. IEEE Softw. 2008. doi:10.1109/ms.2006.113.
- [10] OASIS. Reference Model for Service Oriented Architecture 1.0. OASIS Standard. Public Rev Draft 2. 2006;(October):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, Gallardo G, Hernantes J, Serrano N. DevOps. IEEE Softw. 2016. doi:10.1109/MS.2016.68.
- [13] Lewis J, Fowler M. Microservices. <http://martinfowler.com>. <http://martinfowler.com/articles/microservices.html>. Published 2014.
- [14] Lewis J, Henry S. A methodology for integrating maintainability using software metrics. Conf Softw Maint. 1989. doi:10.1109/ICSM.1989.65191.
- [15] Ferreira KAM, Bigonha MAS, Bigonha RS, Mendes LFO, Almeida HC. Identifying thresholds for object-oriented software metrics. In: Journal of Systems and Software. ; 2012. doi:10.1016/j.jss.2011.05.044.
- [16] Fontana FA, Zanoni M. On investigating code smells correlations. In: Proceedings - 4th IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2011. ; 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) DOI: 10.1109/IEEESTD.2006.235774.
- [18] Gill, G. K., & Kemerer, C. F. (1991). Cyclomatic complexity density and software maintenance productivity. IEEE transactions on software engineering, 17(12), 1284.
- [19] ISO:ISO/IEC 25010:2011, Systems and software engineering --Systems and software Quality Requirements and Evaluation (SQuaRE) --System and software quality models
- [20] Humphrey WS. Characterizing the Software Process: A Maturity Framework. IEEE Softw. 1988. doi:10.1109/52.2014.
- [21] Hynninen T, Kasurinen J, Knutas A, Taipale O. Software testing: Survey of the industry practices. In: 2018 41st International Convention on Information and

Communication Technology, Electronics and Microelectronics, MIPRO 2018 - Proceedings. ; 2018. doi:10.23919/MIPRO.2018.8400261.

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

[23] Jehan S, Pill I, Wotawa F. Functional SOA testing based on constraints. In: 2013 8th International Workshop on Automation of Software Test, AST 2013 - Proceedings. ; 2013:33-39. doi:10.1109/IWAST.2013.6595788.

[24] Ten Dyke RP, Kunz JC. Object-oriented programming. *IBM Syst J*. 2010. doi:10.1147/sj.283.0465.

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

[26] Motogna S, Vescan A, Serban C, Tirban P. An approach to assess maintainability change. In: 2016 20th IEEE International Conference on Automation, Quality and Testing, Robotics, AQTR 2016 - Proceedings. ; 2016. doi:10.1109/AQTR.2016.7501279.

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

[28] Yamashita A. Experiences from performing software quality evaluations via combining benchmark-based metrics analysis, software visualization, and expert assessment. In: 2015 IEEE 31st International Conference on Software Maintenance and Evolution, ICSME 2015 - Proceedings. ; 2015. doi:10.1109/ICSM.2015.7332493.

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

[30] Peffers K, Tuunanen T, Rothenberger MA, Chatterjee S. A Design Science Research Methodology for Information Systems Research. *J Manag Inf Syst*. 2007;24(3):45-77. doi:10.2753/MIS0742-1222240302.

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

[32] Tufano M, Palomba F, Bavota G, et al. When and Why Your Code Starts to Smell Bad - Additional Analysis. In: 37th IEEE/ACM International Conference on Software Engineering. ; 2015.

- [33] Rhoton J, Haukioja R. Cloud Computing Architected. Recursive Press; 2011.
- [34] Kannangara, S. H., & Wijayanayake, W. M. J. I. (2015). An empirical evaluation of impact of refactoring on internal and external measures of code quality IJSEA Journal, Vol.6, No.1, January 2015, PP. 51 - 67
- [35] Kamei Y, Shihab E, Adams B, et al. A large-scale empirical study of just-in-time quality assurance. IEEE Trans Softw Eng. 2013. doi:10.1109/TSE.2012.70.
- [36] Robson C. Real world research. Blackwell Publ. 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 an adjunct professor of entertainment software engineering. His research interests include but are not limited to software testing, quality assurance and software maintenance.

Manuscript received on 28 September 2020

Publication V

Hynninen, T., Kasurinen, J., Knutas, A., and Taipale, O.

Guidelines for software testing education objectives from industry practices with a constructive alignment approach

In Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in
Computer Science Education

Pp. 278–283, 2018

© 2018, ACM

Guidelines for Software Testing Education Objectives from Industry Practices with a Constructive Alignment Approach

Timo Hynninen
Lappeenranta University of
Technology
P.O. Box 20
FI-53851 Finland
timo.hynninen@lut.fi

Jussi Kasurinen
South-Eastern Finland University of
Applied Sciences (XAMK)
Finland
jussi.kasurinen@xamk.fi

Antti Knutas
Lero – The Irish Software Research
Centre
Ireland
antti.knutas@lero.ie

Ossi Taipale
Lappeenranta University of
Technology
P.O. Box 20
FI-53851 Finland
ossi.taipale@lut.fi

ABSTRACT

Testing and quality assurance are characterized as the most expensive tasks in the software life cycle. However, several studies also indicate that the industry could enhance product quality and reduce costs by investing in developing testing practices. Software engineering educators can bridge the gap between formal education and industry practices to produce more industry-ready graduates, by observing the industry in action. To find out the current state of industry, we conducted a study in software organizations to assess how they test their products and which process models they follow. According to the survey results, the organizations rely heavily on test automation and use sophisticated testing infrastructures, apply agile practices even when working with mission-critical software, and have reduced the use of formal process reference and assessment models. Based on the results, this paper identifies a number of key learning objectives in quality assurance and software testing disciplines that the industry expects from university graduates. The principles of constructive alignment are used to present learning goals, teaching methods, and assessment methods that align with the industry requirements.

CCS CONCEPTS

• **Social and professional topics** → Software engineering education; • **Software and its engineering** → Software verification and validation

KEYWORDS

Curriculum, software testing, constructive alignment

ACM Reference format:

Timo Hynninen, Jussi Kasurinen, Antti Knutas and Ossi Taipale. 2018. Guidelines for Software Testing Education Objectives from Industry Practices with a Constructive Alignment Approach. In Proceedings of 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE'18). ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/3197091.3197108>

1 INTRODUCTION

Testing is an important part of software life cycle, as inadequate testing and quality assurance practices in can cause substantial immediate costs as well as poor quality and high maintenance products. Proper testing education can improve software quality, for example students more experienced in testing produce more reliable code [1], [2].

Constructive alignment is an outcomes-based approach to teaching in which the learning outcomes that students are intended to achieve are defined before teaching takes place [3]. Teaching and assessment methods are then designed to best achieve those outcomes and to assess the standard at which they have been achieved. The teaching environment, practices and evaluation should support learning goals and the student's future environment [4].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ITiCSE '18, July 2–4, 2018, Larnaca, Cyprus
© 2018 Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5707-4/18/07...\$15.00
<https://doi.org/10.1145/3197091.3197108>

In order to align the testing education content with industry practices, the following research questions were formulated: 1) Which testing tools and technologies are most used in the industry? 2) What are the current issues related to testing in the industry? 3) How should the learning goals, teaching methods and evaluation methods in a software testing course constructively aligned with current industry practices?

Rest of the paper is structured as follows: Section 2 introduces related work in testing education and testing surveys. Section 3 describes the research process. The survey and its results are presented in Section 4. The constructive alignment model and guidelines are given in Section 5. Finally, we conclude in Section 6.

2 RELATED RESEARCH

There are several studies that implicitly investigate software testing education from a constructive or a requirements alignment perspective without explicitly citing the theory. For example, a study by Krutz et al. [5] investigates motivational issues and placing students into a more real-life like environment that supports learning. The studied issue was that students tend to think testing and quality assurance work as boring and unnecessary extra work. In the Krutz et al. study, the motivational problems were addressed by applying real open-source software projects as the course assignments to give the tasks more realistic scope and scale. Based on their results, 85 percent of their students considered this approach to be positive, with student feedback also indicating improved motivation and learning results. Similar observations were also reported in a study by Garousi and Mathur [6], which also observed that it is not uncommon for a computer science degree program to omit the concept of quality assurance and software testing from their course curricula. In another similar study Broman et al. [7] discuss aligning software testing course with real world practices, and explicitly use the theory of constructive alignment.

Another set of studies create requirements for software testing courses and present ways to align courses with these interventions. A study by Smith et al. [8] discusses the general requirements for developing a testing course: the university course has to be fun and competitive, allow students to learn from each other, the assignments have to demonstrate the importance of doing testing work, and provide an example of the scale and difficulty of the real-world quality assurance issues. They also present an example intervention where a course is changed to align with. These are important considerations, because for example in study aligning course curricula with the games industry [9], it was established that the academia and the industry do not share a common view on what are the necessary and important skills for the students to possess, especially when considering more theoretical topics beyond the set of taught programming languages. In this sense, it would be important to collect information on the tools and strategies applied by the industry, in the development of a course with industry-applicable experience, especially since the more refined testing

tools applied by the industry require domain-related expertise [6] and which may actually be difficult or expensive to acquire without support from the degree program [10].

3 RESEARCH PROCESS

The objective of this study was to align testing education with industry practices and needs. We used surveys as the primary research method to study the industry, as surveys are used to collect information from people about their feelings and beliefs [11]. We consider the constructive alignment approach as an exploratory study for which the survey method is appropriate [12].

We used the questionnaire form introduced in Kasurinen et al. [13] and originally designed in 2005 [14] to get information about the respondents' organization profile, testing practices, test process maturity, applied process models and the tasks related to software development. The questionnaire comprised of eleven chapters containing multi-item, multi-choice questions and open-ended questions. The multi-item questions used a five-point Likert scale (1 fully disagree - 3 neutral - 5 fully agree). The reliability of the multi-item questions in the chapters were originally estimated by using the Cronbach alpha coefficient. In addition to the original 2009 questions, we added new questions about the costs of maintenance and product support.

The sampling method was convenience sampling, with as wide reach as possible within the industry. We advertised the survey in social media platforms such as LinkedIn, Facebook, Twitter and Researchgate and by direct contacts to our industrial partners and open calls for participation in several public online discussion channels. We used advertisement channels to get responses especially from our alumni, and asked them to also share the survey on social media. In order to avoid an extremely biased and small sample anyone working in the software industry was welcome to take part.

The questionnaire collected 33 responses from individuals in working in different organizations. The survey form was opened 930 times (by unique clients or IP-addresses) resulting in a response rate of 3.5 percent which is fairly normal for Internet surveys [11]. To estimate the sample size, we used publicly available statistics provided by the Ministry of Economic Affairs and Employment of Finland [15]. According to the latest estimate, there were 3360 companies in the software business sector, making the sample size approximately 1 percent of the Finnish software industry.

Finally, we used the recommendations for constructive aligned teaching [3], [4] to derive learning goals for industry practices that were collected in the survey. From there a set of teaching methods and the performances of understanding required for evaluation were designed, informed by the same recommendations. They are summarized follows:

- Learning goals should be clear, serve a purpose, and set in advance.
- Students need to be placed in situations and environments that elicit the required learnings, with declarative teaching minimized.

- Students are then required to provide evidence, either by self-set or teacher-set tasks, as appropriate, that their learning can match the stated objectives.

4 SURVEY RESULTS

Questions in the survey addressed the testing and quality assurance practices, the tools used to support these activities as well as development practices and problems.

In terms of organizational profile, very small, small and medium-sized organizations represented each about 21 % of the participants, while 36.4 % were large or very large (more than 250 employee) organizations. Approximately eighty percent of the organizations were private companies, the rest being government or nonprofit organizations. Respondents from organizations focusing primarily on national operations formed 21.2 % of the total while 39.3 % focused mostly on international business. Respondents in 51.5 % of the organizations reported that product fault could cause remarkable economic losses, and 18.2 % considered themselves primarily as open source developers. The majority of the respondents (66.7 %) were primarily software developers, 12.1% were managers and 15.2% worked in quality assurance.

The first chapter of the questionnaire was about the application level of different software testing tools. A tool was defined as “any application, framework, web service, extra library, feature of your development environment etc. whichever supports the activity in question.” The four most popular tool categories include defect reporting, test automation, unit testing and defect/code tracing tools, which are used by over half of all surveyed organizations. Table 1 presents the number of used testing tools.

The second chapter of the questionnaire consisted of multi-choice questions about the severity of test and quality assurance problems. The questions covered topics such as which issues slow down the development, which issues currently restrict testing, and how well current testing tools support development needs. The issues in the questions were originally identified in 2009 [13]. The results indicate that the configurability of the testing tools is a common issue. In addition, feature development in the late phases of development can have an effect on testing schedule, and insufficient communication can slow down defect fixing. Another problem highlighted from the responses was that testing personnel do not have enough expertise in certain testing applications.

The third and fourth chapter of the survey addressed software processes and the amount of agile practices in the organizations. In general, the results indicate that the industry is quite confident in the use of agile practices. The industry drive towards agile can also be observed from the questions concerning the use of formal process models such as SPICE (software process assessment, ISO/IEC 15504, currently part of the ISO/IEC 33000 series) [16] or software testing standard (ISO/IEC 29119) [17]. The questions covered also the utilization of capability and maturity models, such as TMMi - test maturity model integrated [18] or CMMi - capability maturity model

integrated [19]. Some form of process model (formal or self-defined) was applied by only 21.2 percent of organizations, while

Table 1: The percentage of applied testing tools in the industry.

Tool	Percentage of respondents
Bug/Defect reporting	72.7 %
Test automation	66.7 %
Unit testing	57.6 %
Bug/Code tracing	57.6 %
Performance testing	48.5 %
Test case management	45.5 %
Integration testing	45.5 %
Virtual test environment	42.4 %
Quality control	36.4 %
Automated metrics collector	36.4 %
System testing	27.3 %
Security testing	24.2 %
Test completeness	24.2 %
Test design	15.2 %
Protocol/Interface conformance tool	9.1 %

according to the respondents none of the organizations applied capability or maturity certificates. V-model, acceptance criteria for tickets and “generic agile” were mentioned by name, all based on best practices collected from various sources and “self-defined”. No standard, model or certificate program was directly named. Also, in some organizations individual employees are unsure about the application of process models or capability certificates.

The final chapter in our survey included several questions concerning the software testing and quality assurance practices. Respondents were asked to evaluate how well different statements about development practices fit their organizational unit on a scale of 1 (fully disagree) to 5 (fully agree). The statements and survey responses in are presented in Table 2. We present mode as the primary indicator for the individual statements, as the survey used an interval Likert scale. The organizational units are more confident on their system level quality assurance (system, acceptance) testing than on the unit or integration level testing. Organizational units are also confident that they are building the product right, and at the same time, building the right product. Testing schedules may not be kept (mode 2, partially disagree) and time is not necessarily allocated enough for testing (mode 2, partially disagree). Code review practices are varying between different organizations (mode 1, fully disagree).

In addition to multi-choice questions the survey contained open-ended questions, where respondents were asked to explain how their organization manages testing and maintenance related effort. The following themes in managing testing-related work were highlighted from the open responses:

- Moving from proprietary software to open source
- Increasing the coverage of automated tests
- Focusing on service scalability in design
- Re-implementing legacy applications
- Setting up dedicated testing and development environments
- Offshoring testing work
- Establishing pre-planned maintenance time for projects, during which last defects are fixed
- Forming dedicated maintenance teams
- Emphasizing the responsibility of current developers
- Employing a risk-based testing approach to cover the most critical components rather than trying to get perfect coverage.

5 DISCUSSION AND IMPLICATIONS

To answer the first research question, *which testing tools and technologies are most used in the industry*, the most common tools in 2017 were defect-reporting, unit testing and test automation tools. Test case management and test design tools were the categories with decreasing usage. Test automation tools are popular on every level of automation (data collection, performance, general automation and tracing). Automated testing is considered cheap. However, the quality and coverage of testing is a concern to some developers.

In terms of the second research question, *issues related to testing in the industry*, the configurability of testing tools and personnel not being familiar with certain testing tools were common issues according to the survey. Although it is unclear if the respondents meant personnel not being familiar with a particular application their company uses or with tools of a particular type, this result highlights the importance of having students use a variety of tools already during their studies. The test process follows a certain path, executing the test phases regardless of the project limitations. Emphasis is put on the late phases, such as acceptance testing phase. Some form of a systematic process or method in testing is followed by 21.2% of the software companies even though over half of the companies use the most common testing tools.

Interestingly, the static testing practices are very varying between our respondents. While some organizations keep code reviews and go through checklists, about half of the responses say the opposite. One possible explanation for this result may be the fact that there were many respondents from small companies who employ extreme agile development processes and have not yet established formal processes for code reviews, walkthroughs or checklists.

The third research question, *constructively aligning a software testing education*, is addressed next. In Table 3 we present an initial design for a software testing course whose learning goals, teaching methods, and evaluation methods have been constructively aligned based on the industry survey results. In this design we aim to minimize declarative teaching, place students in environments that elicit required learnings on

software testing and evaluate with “performances of understanding,” as recommended in the guidelines by Biggs [3], [4]. It should be noted that the model presented is not exclusive. In other words, we recommend including listed topics in software testing education, but do not recommend excluding any topics that we do not list.

Additionally, we suggest the following guidelines for

Table 2: The self-assessment of the testing and quality assurance practices (1 fully disagree – 3 neutral – 5 fully agree).

Question	Average	Mode
Our software correctly implements a specific function. We are building the product right.	4.1	4
Our software is built traceable to customer requirements. We are building the right product.	3.8	5
Our formal inspections are OK.	3.4	4
We go through checklists.	3.0	2
We keep code reviews.	3.2	1
Our unit testing (modules or procedures) is excellent.	2.9	4
Our integration testing (multiple components together) is excellent.	3.0	3
Our usability testing (adapt software to users' work styles) is excellent.	3.0	3
Our function testing (detect discrepancies between a program's functional specification and its actual behavior) is excellent.	2.9	3
Our system testing (system does not meet requirements specification) is excellent.	3.4	3
Our acceptance testing (users run the system in production) is excellent.	3.6	4
We keep our testing schedules.	3.2	2
Last testing phases are kept regardless of the project deadline.	3.0	4
We allocate enough testing time.	2.6	2

constructive alignment of testing curriculum:

- Incorporate the use of the most common testing tools, defect reporting, unit testing and test automation, into the curriculum. The students will most likely require the skill to use these tools in their future workplace.

Table 3: The constructive alignment of software testing course goals and methods to industry practices.

Learning goals	Teaching methods	Assessment methods ("performances of understanding")
Learn the practice of defect reporting and the use of bug tracking tools	Individual exercises: Find and report bugs.	Demonstrate understanding through the individual projects
Implementing unit tests and evaluating test coverage	Individual exercises: Create a program and set up unit tests	
Independent implementation of test automation	Individual exercises: Set up full testing automation for a program	
Understand and apply test process design in future projects	Teamwork: Project management exercise and testing process simulation	Demonstrate understanding through equal contribution to the teamwork project (individual and group evaluation)
Integrating testing phases to software engineering practices	Teamwork: Project management exercise; acceptance testing between two teams	
Evaluating and managing technical debt; making rational compromises	Teacher-led exercise: A review of the shortcuts taken during the course, and discussion & evaluation of the long-term drawbacks of the shortcuts	Demonstrate understanding by a written assignment that reviews and evaluates technical issues
Implementing static testing: Creating checklists and performing code reviews	Teamwork: Going through checklists and reviewing each other's code. TA acts as QA manager in final projects	Demonstrate understanding by working in a simulated verification and validation review

- Use popular, widely used testing tools rather than tools designed for education, in order to teach students the correct use and configuration of real environments.
- Emphasize the importance of static testing methods as the way to improve code quality.
- Produce documentation early on to encourage a mindset for documenting the progress of the project.
- Use a variety of tools for the same purpose to give students experience of the different tools available.
- Enforce documentation practices to enhance the communication skills, for example producing and handling defect reports.

The ACM computer science curricula places testing skills in the knowledge area of software development fundamentals. Verifying program correctness is an extensive topic in the core contents of the recommendation. Testing activities in the ACM software engineering curricula are mainly under the Software Verification and Validation knowledge area, although testing themes span across multiple areas of knowledge such as Software process or Software quality. Although the ACM curricula recommendations cover testing well, they have been criticized for not providing students a rigorous enough testing mindset [20].

6 CONCLUSIONS

In this paper, we presented the alignment of software testing education goals to industry practices. We observed the industry by conducting a survey on testing tools and quality assurance practices. The survey results indicated a strong preference

towards agile development practices and high use of automation. Moreover, the use of formal process reference and assessment models was in the minority. In addition, the survey results ranked the popularity of different testing tools, which directly benefits the software engineering educators.

The survey results were used to constructively align software testing education with industry practices and expectations, producing a course model that responds to industry needs. The presented model can be used as a frame of reference for the learning objectives related to testing work in computer science education. Additionally, a number of guidelines for actual course content were presented.

The study addressed a similar issue as in Krutz et al. [5] and Broman et al. [7], though from a different perspective. We took a step back and gather requirements and learning objectives for a course on software testing, rather than investigate how the requirements can be used to constructively align a course. This approach is similar to the work of Garousi and Mathur [6] who performed a review as well, though they surveyed existing degree programs instead of the industry.

The limitations of the study warrant some discussion. The sampling of our survey was limited to a one country, and for this reason the results are not strong and confirmatory. However, we consider the survey results as exploratory from which estimates can be drawn.

In future work the actual learning activities and course organization should be addressed. One topic of interest could be the alignment of actual software testing activities with the different phases of software life cycle.

ACKNOWLEDGMENTS

This work was partially funded by the Technology Development center of Finland (TEKES), as part of the .Maintain project

(project number 1204/31/2016). The work of the third author was supported by the Ulla Tuominen foundation.

REFERENCES

- [1] O. A. L. Lazzarini Lemos, C. Cutigi Ferrari, F. Fagundes Silveira, and A. Garcia, "Experience report: Can software testing education lead to more reliable code?," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, 2015, pp. 359–369.
- [2] O. A. Lazzarini Lemos, F. Fagundes Silveira, F. Cutigi Ferrari, and A. Garcia, "The impact of Software Testing education on code reliability: An empirical assessment," *Journal of Systems and Software*, Mar. 2017.
- [3] J. Biggs, "Constructive alignment in university teaching," *HERDSA Review of Higher Education*, vol. 1, no. 5, pp. 5–22, 2014.
- [4] J. Biggs, "Enhancing teaching through constructive alignment," *Higher education*, vol. 32, no. 3, pp. 347–364, 1996.
- [5] D. E. Krutz, S. A. Malachowsky, and T. Reichlmayr, "Using a Real World Project in a Software Testing Course," in *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, New York, NY, USA, 2014, pp. 49–54.
- [6] V. Garousi and A. Mathur, "Current State of the Software Testing Education in North American Academia and Some Recommendations for the New Educators," in *2010 23rd IEEE Conference on Software Engineering Education and Training*, 2010, pp. 89–96.
- [7] D. Broman, K. Sandahl, and M. A. Baker, "The company approach to software engineering project courses," *IEEE Transactions on Education*, vol. 55, no. 4, pp. 445–452, 2012.
- [8] J. Smith, J. Tessler, E. Kramer, and C. Lin, "Using Peer Review to Teach Software Testing," in *Proceedings of the Ninth Annual International Conference on International Computing Education Research*, New York, NY, USA, 2012, pp. 93–98.
- [9] M. M. McGill, "Defining the Expectation Gap: A Comparison of Industry Needs and Existing Game Development Curriculum," in *Proceedings of the 4th International Conference on Foundations of Digital Games*, New York, NY, USA, 2009, pp. 129–136.
- [10] F. Kazemian and T. Howles, "A Software Testing Course for Computer Science Majors," *SIGCSE Bull.*, vol. 37, no. 4, pp. 50–53, Dec. 2005.
- [11] A. Fink, *How to Conduct Surveys: A Step-by-Step Guide*. Sage Publications, 2012.
- [12] B. A. Kitchenham *et al.*, "Preliminary guidelines for empirical research in software engineering," *IEEE Transactions on software engineering*, vol. 28, no. 8, pp. 721–734, 2002.
- [13] J. Kasurinen, O. Taipale, and K. Smolander, "Software test automation in practice: empirical observations," *Advances in Software Engineering*, vol. 2010, 2010.
- [14] O. Taipale, K. Smolander, and H. Kälviäinen, "Finding and Ranking Research Directions for Software Testing," in *Software Process Improvement: 12th European Conference, EuroSPI 2005, Budapest, Hungary, November 9–11, 2005. Proceedings*, I. Richardson, P. Abrahamsson, and R. Messnarz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 39–48.
- [15] T. Metsä-Tokila, "Kasvun mahdollistajat - toimialaraportti ohjelmistoalasta ja teknisestä konsultoinnista" [Enablers of growth – sector report on the software industry and technical consulting], Työ- ja elinkeinoministeriö [Ministry of Economic Affairs and Employment], 2014. [Online]. Available: <http://urn.fi/URN:NBN:fi-fe2017102550274>. [Accessed: 4-Apr-2018].
- [16] "ISO/IEC 15504-1: Information technology – Process assessment – Part 1: Concepts and vocabulary," International Organization for Standardization, 2004.
- [17] ISO/IEC, "ISO/IEC 29119-1 Software and systems engineering - Software testing - Part 1: Concepts and definitions." 2013.
- [18] E. van Veenendaal and B. Wells, *Test Maturity Model Integration TMMi*. The Netherlands: Uitgeverij Tutein Nolthenius, 2012.
- [19] R. Kneuper, *CMMI: Improving Software and Systems Development Processes Using Capability Maturity Model Integration*. Rocky Nook, 2008.
- [20] P. H. D. Valle, E. F. Barbosa, and J. C. Maldonado, "CS curricula of the most relevant universities in Brazil and abroad: Perspective of software testing education," in *Computers in Education (SIE), 2015 International Symposium on*, 2015, pp. 62–68.

Publication VI

Hynninen, T., Knutas, A., and Kasurinen, J.

Designing Early Testing Course Curricula with Activities Matching the V-Model Phases

In Proceedings of the 42nd International Convention on Information and
Communication Technology, Electronics and Microelectronics (MIPRO)

Pp. 1593–1598, 2019

© 2019, IEEE

Designing Early Testing Course Curricula with Activities Matching the V-Model Phases

Timo Hynninen*, Antti Knutas** and Jussi Kasurinen**

*South-Eastern Finland University of Applied Sciences / Department of Information Technology, Mikkeli, Finland

**LUT University / LUT School of Engineering Science, Lappeenranta, Finland

timo.hynninen@xamk.fi, antti.knutas@lut.fi, jussi.kasurinen@lut.fi

Abstract—This work addresses the gap between software engineering process terminology in formal education, and the practical skills relevant to testing related work. The V-model is a commonly referenced description of how the software engineering processes are tied to the different software testing levels. It is used in software engineering education to illustrate which type of testing work should be carried out during a certain development stage. However, the V-model is mainly conceptual and tied to the steps in the Waterfall model, leaving the students with little knowledge about what is actually done. To solve this problem, we propose an approach to map the V-Model development phases and testing levels with corresponding, actual testing techniques. We then evaluate the approach by designing the weekly topics, learning goals and testing activities for a 7 week introductory course on the basics software testing and quality assurance. Based on the course outcomes and recent literature, we discuss the strengths and weaknesses of the proposed curriculum.

I. INTRODUCTION

Software testing and quality assurance (QA) form an integral part of software engineering processes and therefore should be an equally integral part of software engineering education. Testing education improves software quality, as testing-savvy students learn techniques that lead to more reliable program code [1]. For example, the ACM curricula for software engineering [2] integrates testing and quality assurance into other domains of computing education. However, the ACM curricula has been criticized for not conveying a strong enough testing and quality assurance mindset [3]. Our proposal to address this issue is to design concrete learning objectives and testing activities that follow the principles of constructive alignment [4], [5]. This approach carries over from our previous work in using constructive alignment, creating high-level guidelines for testing education from the industry practices [6].

The motivation behind designing a dedicated undergraduate testing course and its activities was that currently testing education research has mainly focused on the implementations of such courses and not in course content design. Although previous research has also established approaches to integrating testing and QA work into larger projects [7], [8], many institutions organize an undergraduate course in the methods and models of software testing separately. In addition, the software industry leaves a lot of responsibility in QA work to the shoulders of individual

employees, while acknowledging that personnel do not always have the necessary skills in testing beforehand [9]. We therefore feel that the objectives of this study are of interest to many in higher education.

In order to place the testing activities into a software engineering context, we contrast them with the phases in the V-Model [10], [11]. The V-Model (see Figure 1) is a generic software development process model where requirement analysis, specification, architectural design, and detail design are linked with the levels of testing, namely acceptance testing, system testing, integration testing, and unit testing. These development process phases and testing levels are often referenced in software engineering education. However, in the education and training context, the practical impact of these activities may play an auxiliary role or even be neglected. Hence, students might be familiar with the development process phases on an abstract level but fail to understand which practical activities should happen within them.

To summarize, our research questions in this paper are as follows.

- RQ1. What learning activities can we map to the high-level testing concepts?
- RQ2. Which actual testing techniques can be utilized?
- RQ3. How do these activities and high-level concepts relate to other software engineering processes, namely the V-Model activities?

In order to answer the research questions, we used the principles of constructive alignment [4], [5] to design and implement an undergraduate course on the fundamentals of software testing. We planned the topics and activities for a 7-week (one period), first-year freshman course. The course had no other prerequisites except the freshman course on introductory programming. Various techniques for testing were adapted from the ISO/IEC Software testing standard, which covers a multitude of testing techniques in ISO/IEC 29119 Part 4 [12]. These testing techniques were used as a starting point for designing assignments demonstrating the practical testing work on each testing level.

The rest of this paper is organized as follows. Recent studies on testing education are presented in Section 2. Section 3 introduces our course implementation and its results. Discussion and implications are discussed in

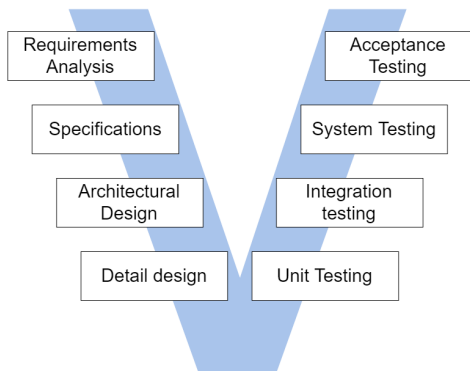


Fig. 1. The software testing V-Model, adapted from [11]

Section 4. Finally, we conclude in Section 6.

II. RELATED WORK

Educators face challenges when it comes to testing concepts. For example, students find it difficult to digest testing concepts unless they are introduced carefully [13]. In addition, many instructors do not have the necessary knowledge that should be taught to students [1]. Additionally, the motivational aspects especially in technically challenging topics are well-known factors influencing the outcomes of a learning scenarios to a significant degree [14].

Testing course content has been studied in different countries, for example by ŠošiĆ in Serbia [15], Bin in China [16] and Kasurinen in Finland [17]. According to Kasurinen, students want testing education to be practice oriented, using real-world tools with a real software project to promote the motivational aspects of learning something practical, which provides skills applicable in real-life software development work.

Experiences from running traditional university courses in software testing, or software verification and validation, have been previously reported by Mishra et al. [18] and Lopez et al. [19]. Van Eijck et al. [20] designed a flipped classroom version of the testing course in addition to bundling the testing education around Microsoft's developer software [21].

Gamification in education is a rising trend in the testing field. Fraser [22], Valle [23], Fu [24], and Soska [25] all present recent approaches for gamifying software testing education. In general, gamification can be used to increase student motivation and communication.

Other approaches for increasing student motivation on the testing course include using large, real-world projects. For example, Krutz et al. [7] used open-source projects and Garousi industrial software projects [8]. Another interesting approach employed by Chen et al. infused research topics into the testing curriculum [26].

III. COURSE IMPLEMENTATION

In the academic year 2017-2018 the course Principles of Software Testing was arranged in parallel with a basic course on C programming. The course population consisted of first year computer science students, and also students majoring in other technology programmes. The objective of the course was to cover the most common software testing methods, give the students an overview of how testing and software engineering are related, and give the students the transferable skills to perform testing related work autonomously or as part of an organization.

We created the course syllabus by taking the high-level objectives and relating the required testing-related skills to the ones that can be acquired by mastering the V-Model. During the process we also mapped the development phases and testing levels from the V-Model to the set of weekly lecture topics on software testing practices.

During the mapping process we used the theory of constructive alignment as the guiding principle when setting learning goals and designing course activities. Constructive alignment is an outcomes-based approach to teaching in which the learning outcomes that students are intended to achieve are defined before teaching takes place [5]. Teaching and assessment methods are then designed to best achieve those outcomes and to assess the standard at which they have been achieved. The teaching environment, practices and evaluation should support learning goals and the student's future environment [4]. We summarize the principles of constructive alignment [4], [5] as follows:

- Learning goals should be clear, serve a purpose, and set in advance.
- Students need to be placed in situations and environments that elicit the required learnings, with declarative teaching minimized.
- Students are then required to provide evidence, either by self-set or teacher-set tasks, as appropriate, that their learning can match the stated objectives.

In the next phase, we took different testing activities and test techniques, and placed them under the weekly lecture schedule. The testing techniques were taken from the ISO/IEC 29119 Software testing standard. The different testing activities were carefully selected to fit the development phase and test levels according to the V-Model activities. For example, Black-box testing and exploratory testing techniques were used as exercises on the system testing level, whereas the unit testing level used the White-box testing approach. Similarly, state transition testing, scenario testing and random testing were used as the approach to specification and requirement analysis, while the classification tree method was used on the integration testing level.

In total, the course consisted of seven weeks of instruction in the form of lectures and voluntary exercise (tutoring) sessions. The weekly course topics are presented in Table I.

The concepts and skills covered in the course material were assessed in two parts: First, the students performed,

TABLE I
THE WEEKLY ACTIVITIES, COVERED TOPICS, AND TESTING TECHNIQUES

Week	Development phase (V-Model)	Test level (V-Model)	Weekly covered topic(s)	Activities and testing techniques applied in them	Learning goals
1	Specification and requirement analysis	System testing	Introduction to testing. Objectives of testing	Black-box system testing. Exploratory testing. Boundary value analysis. Defect reporting.	Understand the objectives of testing work. Student is able to create (Black-box) test cases. Student understands the scope, and limitations of the black box methods.
2	Detail design	Unit testing	Testing levels. Unit testing	White-box testing. Test case reporting. Equivalence partitioning.	Understand the concept of unit / module test. Understand the difference between Black-box and White-box testing.
3	Architectural design	Integration testing	Integration testing	Combinatorial methods and the classification tree method. Test stubs.	Understand the infeasibility of "testing everything." Student is able to select a technique for deriving test cases. Student understands the scope, and limitations of the software testing in the real world software projects.
4	Specification and requirement analysis	System testing. Acceptance testing	System testing	State transition testing. Scenario testing. Random testing.	Understand the objectives of system-level testing. Student is able to select an appropriate testing technique for system testing. Student understands the scope, and limitations of the system-level testing methods.
5	Detail design	Unit testing	Test automation and tools	Implementing unit tests in code, using a unit testing framework	Student is able to use a programming framework / library to implement module tests. Student understands the scope, and limitations of the unit testing tools.
6	Architectural design	System testing	Testing processes, documentation and planning	Creating test plans. Code review and static testing methods. Test coverage analysis.	Student understands the purpose of static testing methods and code review practices.
7	Specification, architectural design, detail design	System, integration, unit testing	Visiting lecture from a software company	Course project: Plan, design, implement and document testing for a small software item.	Student is able to demonstrate their knowledge by applying the course's activities autonomously in the testing project. Student is able to explain how test process activities would relate to the whole software project.

reported and planned a small-scale testing project using a small console application and its specification. The assignment was completed in groups of three and it accounted for 35% of the total course grade. A written exam worth 25% of the grade formed the second part of course assessment. Voluntary weekly exercises, also completed in groups, formed the rest of the course grade, but the emphasis in grading was on the project and the exam.

The testing project was graded by the head teaching assistant based on the completion of each of the individual five parts. Parts 1 and 2 consisted of system testing activities. First, the tests were completed manually using exploratory testing as the main method. Then in part 2 the assignment was to automate some of the test cases developed in part 1 by recording the inputs and program outputs during the test.

In part 3 of the project we tasked the students with writing unit tests for individual modules of the software. Part 4 was an exercise in testing work from a managerial

point of view, and students were to develop a testing plan for the project software as if it was a real product by a real software company. Part 5 consisted of reporting the whole project and documenting in the write-up which test cases they had developed, which were automated, and what unit tests were added to the project repository. Additionally, the report included testing logs and bug reports from the manual system testing phase.

In the final project students were given free choice of tools. Additionally, the problem description did not specify which testing methods or approaches should be used in the different parts as one objective of the project was that students select a suitable method and justify it in the test plan. The various testing techniques had been covered previously in the weekly exercises, where tools for unit testing and test coverage were also introduced.

The final exam consisted of two essay questions about the concepts presented in the lecture material. The exam was graded by the lecturer.

Descriptive course statistics are presented in Table II.

TABLE II
DESCRIPTIVE STATISTICS FROM THE INTRODUCTORY TESTING
COURSE

Students working in the course	124
Group projects returned	43
Average project grade (median)	3.4 (4)
Average exam grade (median)	3.8 (4.5)
Respondents in the post-course survey (%)	19 (15%)
"The course implementation helped me to achieve the learning outcomes of the course" (1 - very poorly, 3 - neutral, 5 - very well)	3.33
"The teaching methods used on the course supported my learning" (1 - very poorly, 3 - neutral, 5 - very well)	3.39

TABLE III
THE MOST COMMON TYPES OF PROJECT FEEDBACK GIVEN TO
STUDENTS BY THE TA

Unit tests did not check that the functions manipulated data correctly, only that their return value reported 'success'	49% (21)
Manual system testing was comprehensive	37% (16)
Objectives of the testing project were unclear or undefined	35% (15)
Unit tests were implemented without the use of a testing framework. The results of the tests were often presented in a way which required the tester to verify the results manually.	16% (7)
The tests seemed to only concentrate on crashing the program using only bad/sketchy inputs.	14% (6)
Unit tests were comprehensive, and tested the actual data manipulation	5% (2)
Amount of generally positive feedback comments given	37% (16)
Amount of generally negative feedback comments given	63% (27)

In the end a total of 124 students worked actively on the course assignments. The average project grade was 3.4 and the average exam grade 3.8 (on a scale of 1-5 in passing grades). In addition to the course deliverables, a post-course survey was also conducted. Unfortunately, even though the survey got a 15% response rate in relation to the course population, the number of actual respondents remained low, and only 10 students gave written feedback.

Observations from the student testing projects are summarized in Table III. The themes listed were collected from the written feedback on the project given by the teaching assistant to the students. Overall, 37% of the comments in the feedback were positive, and 63% negative pointing out flaws in the implementation or clear misconceptions in the report.

IV. DISCUSSION AND IMPLICATIONS

In assessing the proposed curricula, it is necessary to highlight the importance of an early testing course. Some approaches to testing education emphasize using big, real world projects as the basis to prepare graduates for work in the industry (for example [7], [8]). However, as students first go work in the industry as early as 1.5 years into their studies [27], this approach can be difficult to employ early on.

The ACM Curricula Recommendation for undergraduate software engineering programmes infuses software verification and validation into larger projects and courses [2]. We take a step back and ensure the students have familiarized themselves with testing, verification and validation on the dedicated testing course. Our approach to teaching the testing discipline complements the approach of the ACM Curricula Recommendation: Once the students have acquired the appropriate testing mindset the testing skills can be used in other software engineering and computer science projects.

Next, we summarize the findings from our course by addressing the individual research questions. To answer RQ1, what learning activities can we map to the high-level testing concepts, it is possible to complete activities on all testing levels. The activities can vary from black-box to white-box testing, system testing to unit testing, or something in between.

For the second research question, which actual testing techniques can be utilized, we created weekly assignments for students employing a variety of different techniques taken from relevant literacy, such as Swebok [28] or the ISO/IEC Software testing standard [12]. We incorporated a number of different techniques for deriving test cases on different levels, and additionally covering static testing methods and code reviews. In summary we used exploratory testing, equivalence partitioning, boundary value analysis, combinatorial methods, state transition testing, scenario testing, random testing and static testing in conjunction with unit testing, integration testing, system testing, writing test stubs and drivers, and analyzing test coverage.

Finally, for the third and final research question, how do these activities and high-level concepts relate to other software engineering processes, we can say that our seven week course content is aligned with the V-model. This in our opinion makes it easier for students to grasp how the software engineering processes presented in theory relate to work with real projects, and bridges the gap between formal software engineering terminology and the real world.

However, there are still some issues which need addressing in our course arrangements. The project as a demonstration of learning worked generally well. Especially in the first part of the project, exploratory system testing, the project reports presented testing comprehensively. Students were able to use the different techniques and approaches combined with intuition and creativity to sufficiently cover possible errors. Even if the reports did not directly name a particular method which was used in order to arrive to the test cases, it appears that the students were either formally or informally adequately familiar with these techniques.

The other parts of the project proved to be more challenging for the students. For example in the second part nearly all groups successfully employed an automated system testing pipeline, but it was unclear what the students had set as the objectives for automated testing.

In most cases the students had simply recorded test cases which they knew would fail, resulting in nearly all tests failing.

We can see that the problems revolve around discovered with aligning the objectives and implementations of testing. In part 4 of the assignment, drafting a testing plan and test reports, the actual testing objectives were either shallow or completely neglected. In conclusion, it seems that to the students testing meant finding errors and making the program crash - and not ensuring that the program works. In this sense, it can be argued that the students understood the concept of testing work itself as defined by Myers [29], but not the concept of quality assurance or quality control practices as defined by Kaner et al. [30]. However, testing levels were often referenced in the reports, meaning that students were able to place the testing work within the V-model.

V. CONCLUSIONS

The objective of this study was to map high-level learning objectives into concrete testing activities, and to ground the testing activities firmly into the software engineering processes by using the V-Model as a starting point. In order to accomplish this objective we designed an introductory software testing course, and using the principles of constructive alignment, mapped learning goals to actual weekly activities and testing techniques.

We assessed the course curriculum by examining the outcomes of our seven week testing course. Student's practical assignments were used as demonstrations of learning. We observed from the projects that students were able to adopt the testing mindset and carry out comprehensive and systematic testing on the system testing level. On the other hand, this systematic approach to testing work was mainly carried out on the system level, while many projects had problems with unit tests, integration tests and reporting of the project.

The limitations of the study and the validity of the results warrant some discussion. The assignment reports written as group work are of course not the perfect instrument to measure the learning of individual students. However, as the assignment was split into multiple sub-sections with each section focusing on individual activities (exploratory testing, system testing, unit testing, test planning and documentation), it was easy to see which concepts the students excelled in or struggled with.

Additionally, the lecture material and the reference book used in the course, will most certainly have had a significant impact on the student's perception of testing as a whole. If the reference material is biased towards some topics or does not cover some concept well enough, it can be expected that the student reports follow the same shortcomings. In our case, the course material covers all the concepts expected in the assignments. On the other hand, due to the practical limitations some content discussing more advanced topics had to be discussed only on a high level.

As future work, one promising approach would be incorporating a knowledge acquisition measurement algo-

rithm such as ACT-R or BKT to assess the student performance and learning during the course in order to establish which course components require more refinement. Other prospective area of interest would be the integration of the advanced topics, and including a larger project work, which integrates both the software engineering and software testing methods into one capstone assignment.

REFERENCES

- [1] O. A. L. Lemos, F. F. Silveira, F. C. Ferrari, and A. Garcia, "The impact of software testing education on code reliability: An empirical assessment," *Journal of Systems and Software*, vol. 137, pp. 497–511, 2018.
- [2] The Joint Task Force on Computing Curricula, "Curriculum guidelines for undergraduate degree programs in software engineering," New York, NY, USA, Tech. Rep., 2015.
- [3] P. H. D. Valle, E. F. Barbosa, and J. C. Maldonado, "Cs curricula of the most relevant universities in brazil and abroad: Perspective of software testing education," in *Computers in Education (SIIE), 2015 International Symposium on*. IEEE, 2015, pp. 62–68.
- [4] J. Biggs, "Enhancing teaching through constructive alignment," *Higher education*, vol. 32, no. 3, pp. 347–364, 1996.
- [5] —, "Constructive alignment in university teaching," *HERDSA Review of higher education*, vol. 1, no. 1, pp. 5–22, 2014.
- [6] T. Hynninen, J. Kasurinen, A. Knutas, and O. Taipale, "Guidelines for software testing education objectives from industry practices with a constructive alignment approach," in *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. ACM, 2018, pp. 278–283.
- [7] D. E. Krutz, S. A. Malachowsky, and T. Reichlmayr, "Using a real world project in a software testing course," in *Proceedings of the 45th ACM technical symposium on Computer science education*. ACM, 2014, pp. 49–54.
- [8] V. Garousi, "Incorporating real-world industrial testing projects in software testing courses: Opportunities, challenges, and lessons learned," in *Software Engineering Education and Training (CSEE&T), 2011 24th IEEE-CS Conference on*. IEEE, 2011, pp. 396–400.
- [9] T. Hynninen, J. Kasurinen, A. Knutas, and O. Taipale, "Software testing: Survey of the industry practices," in *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, 2018, pp. 1449–1454.
- [10] P. Rook, "Controlling software projects," *Software Engineering Journal*, vol. 1, no. 1, pp. 7–16, 1986.
- [11] S. Mathur and S. Malik, "Advancements in the v-model," *International Journal of Computer Applications*, vol. 1, no. 12, 2010.
- [12] "Software and systems engineering – software testing – part 4: Test techniques," International Organization for Standardization, Geneva, CH, Standard, 2015.
- [13] D. Mishra, S. Ostrovska, and T. Hacaloglu, "Exploring and expanding students' success in software testing," *Information Technology & People*, vol. 30, no. 4, pp. 927–945, 2017.
- [14] M. Gagné and E. L. Deci, "Self-determination theory and work motivation," *Journal of Organizational behavior*, vol. 26, no. 4, pp. 331–362, 2005.
- [15] S. Šoćić, O. Ristić, K. Mitrović, and D. Milošević, "Software testing course in it undergraduate education in serbia," *Information Technology*, vol. 4, no. 6, p. 8, 2018.
- [16] Z. Bin and Z. Shiming, "Curriculum reform and practice of software testing," in *International Conference on Education Technology and Information System (ICETIS 2013)*, 2013, pp. 841–844.
- [17] J. Kasurinen, "Experiences from a web-based course in software testing and quality assurance," *International Journal of Computer Applications*, vol. 166, no. 2, 2017.
- [18] D. Mishra, T. Hacaloglu, and A. Mishra, "Teaching software verification and validation course: A case study," *International Journal of Engineering Education*, vol. 30, pp. 1476–1485, 2014.
- [19] G. Lopez, F. Cocozza, A. Martinez, and M. Jenkins, "Design and implementation of a software testing training course," in *122nd ASEE Annual Conference & Exposition*, 2015.
- [20] J. van Eijck, V. Zaytsev et al., "Flipped graduate classroom in a haskell-based software testing course," in *Pre-proceedings of the Third International Workshop on Trends in Functional Programming in Education (TFPIE 2014)*, 2014.

- [21] G. Lopez and A. Martinez, "Use of microsoft testing tools to teach software testing: An experience re-port," in *Proceedings of the American Society for Engineering Education Annual Conference and Exposition*, 2014.
- [22] G. Fraser, A. Gambi, and J. M. Rojas, "A preliminary report on gamifying a software testing course with the code defenders testing game," in *Proceedings of the 3rd European Conference of Software Engineering Education*. ACM, 2018, pp. 50–54.
- [23] P. H. D. Valle, A. M. Toda, E. F. Barbosa, and J. C. Maldonado, "Educational games: A contribution to software testing education," in *Frontiers in Education Conference (FIE)*. IEEE, 2017, pp. 1–8.
- [24] Y. Fu and P. Clarke, "Gamification based cyber enabled learning environment of software testing," *submitted to the 123rd American Society for Engineering Education (ASEE)-Software Engineering Constituent*, 2016.
- [25] A. Soska, J. Mottok, and C. Wolff, "An experimental card game for software testing: Development, design and evaluation of a physical card game to deepen the knowledge of students in academic software testing education," in *Global Engineering Education Conference (EDUCON), 2016 IEEE*. IEEE, 2016, pp. 576–584.
- [26] Z. Chen, A. Memon, and B. Luo, "Combining research and education of software testing: a preliminary study," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, 2014, pp. 1179–1180.
- [27] The Joint Task Force on Computing Curricula, "Curriculum guidelines for baccalaureate degree programs in information technology," New York, NY, USA, Tech. Rep., 2017.
- [28] P. Bourque, R. E. Fairley *et al.*, *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0*. IEEE Computer Society Press, 2014.
- [29] G. J. Myers, T. Badgett, T. M. Thomas, and C. Sandler, *The art of software testing*, 2nd ed. John Wiley & Sons, 2004.
- [30] C. Kaner, J. Bach, and B. Pettichord, *Lessons learned in software testing: a context-driven approach*. Wiley, 2002.

ACTA UNIVERSITATIS LAPPEENRANTAENSIS

- 1032. ZOLOTAREV, FEDOR. Computer vision for virtual sawing and timber tracing. 2022. Diss.
- 1033. NEPOVINNYKH, EKATERINA. Automatic image-based re-identification of ringed seals. 2022. Diss.
- 1034. ARAYA GÓMEZ, Natalia Andrea. Sustainable management of water and tailings in the mining industry. 2022. Diss.
- 1035. YAHYA, MANAL. Augmented reality based on human needs. 2022. Diss.
- 1036. KARUPPANNAN GOPALRAJ, SANKAR. Impacts of recycling carbon fibre and glass fibre as sustainable raw materials for thermosetting composites. 2022. Diss.
- 1037. UDOKWU, CHIBUZOR JOSEPH. A modelling approach for building blockchain applications that enables trustable inter-organizational collaborations. 2022. Diss.
- 1038. INGMAN, JONNY. Evaluation of failure mechanisms in electronics using X-ray imaging. 2022. Diss.
- 1039. LIPIÄINEN, SATU. The role of the forest industry in mitigating global change: towards energy efficient and low-carbon operation. 2022. Diss.
- 1040. AFKHAMI, SHAHRIAR. Laser powder-bed fusion of steels: case studies on microstructures, mechanical properties, and notch-load interactions. 2022. Diss.
- 1041. SHEVELEVA, NADEZHDA. NMR studies of functionalized peptide dendrimers. 2022. Diss.
- 1042. SOUSA DE SENA, ARTHUR. Intelligent reflecting surfaces and advanced multiple access techniques for multi-antenna wireless communication systems. 2022. Diss.
- 1043. MOLINARI, ANDREA. Integration between eLearning platforms and information systems: a new generation of tools for virtual communities. 2022. Diss.
- 1044. AGHAJANIAN, SOHEIL. Reactive crystallisation studies of CaCO₃ processing via a CO₂ capture process: real-time crystallisation monitoring, fault detection, and hydrodynamic modelling. 2022. Diss.
- 1045. RYYNÄNEN, MARKO. A forecasting model of packaging costs: case plain packaging. 2022. Diss.
- 1046. MAILAGAHA KUMBURE, MAHINDA. Novel fuzzy k-nearest neighbor methods for effective classification and regression. 2022. Diss.
- 1047. RUMKY, JANNATUL. Valorization of sludge materials after chemical and electrochemical treatment. 2022. Diss.
- 1048. KARJUNEN, HANNU. Analysis and design of carbon dioxide utilization systems and infrastructures. 2022. Diss.
- 1049. VEHEMAANPERÄ, PAULA. Dissolution of magnetite and hematite in acid mixtures. 2022. Diss.
- 1050. GOLOVLEVA, MARIA. Numerical simulations of defect modeling in semiconductor radiation detectors. 2022. Diss.

- 1051.** TREVES, LUKE. A connected future: The influence of the Internet of Things on business models and their innovation. 2022. Diss.
- 1052.** TSERING, TENZIN. Research advancements and future needs of microplastic analytics: microplastics in the shore sediment of the freshwater sources of the Indian Himalaya. 2022. Diss.
- 1053.** HOSEINPUR, FARHOOD. Towards security and resource efficiency in fog computing networks. 2022. Diss.
- 1054.** MAKSIMOV, PAVEL. Methanol synthesis via CO₂ hydrogenation in a periodically operated multifunctional reactor. 2022. Diss.
- 1055.** LIPIÄINEN, KALLE. Fatigue performance and the effect of manufacturing quality on uncoated and hot-dip galvanized ultra-high-strength steel laser cut edges. 2022. Diss.
- 1056.** MONTONEN, JAN-HENRI. Modeling and system analysis of electrically driven mechatronic systems. 2022. Diss.
- 1057.** HAVUKAINEN, MINNA. Global climate as a commons — from decision making to climate actions in least developed countries. 2022. Diss.
- 1058.** KHAN, MUSHAROF. Environmental impacts of the utilisation of challenging plastic-containing waste. 2022. Diss.
- 1059.** RINTALA, VILLE. Coupling Monte Carlo neutronics with thermal hydraulics and fuel thermo-mechanics. 2022. Diss.
- 1060.** LÄHDEAHO, OSKARI. Competitiveness through sustainability: Drivers for logistics industry transformation. 2022. Diss.
- 1061.** ESKOLA, ROOPE. Value creation in manufacturing industry based on the simulation. 2022. Diss.
- 1062.** MAKARAVA, IRYNA. Electrochemical recovery of rare-earth elements from NdFeB magnets. 2022. Diss.
- 1063.** LUHAS, JUKKA. The interconnections of lock-in mechanisms in the forest-based bioeconomy transition towards sustainability. 2022. Diss.
- 1064.** QIN, GUODONG. Research on key technologies of snake arm maintainers in extreme environments. 2022. Diss.
- 1065.** TAMMINEN, JUSSI. Fast contact copper extraction. 2022. Diss.
- 1066.** JANTUNEN, NIKLAS. Development of liquid–liquid extraction processes for concentrated hydrometallurgical solutions. 2023. Diss.
- 1067.** GULAGI, ASHISH. South Asia's Energy [R]evolution – Transition towards defossilised power systems by 2050 with special focus on India. 2023. Diss.
- 1068.** OBREZKOV LEONID. Development of continuum beam elements for the Achilles tendon modeling. 2023. Diss.
- 1069.** KASEVA, JANNE. Assessing the climate resilience of plant-soil systems through response diversity. 2023. Diss.



ISBN 978-952-335-922-2
ISBN 978-952-335-923-9 (PDF)
ISSN 1456-4491 (Print)
ISSN 2814-5518 (Online)
Lappeenranta 2023