Proactive Software Complexity Assessment

Vard Antinyan

Department of Computer Science and Engineering

UNIVERSITY OF GOTHENBURG

Gothenburg 2017

PhD Thesis
Proactive Software Complexity Assessment

© Vard Antinyan 2017

Technical Report No 143D
ISBN 978-91-982237-2-9
Department of Computer Science and Engineering
Division of Software Engineering
University of Gothenburg | Chalmers University of Technology
Printed by Chalmers Reproservice
Gothenburg, Sweden 2017

"Complexity, I would assert, is the biggest factor involved in anything having to do with the software field. It is explosive, far reaching, and massive in its scope".

Robert Glass

ABSTRACT

Large software development companies primarily deliver value to their customers by continuously enhancing the functionality of their products. Continuously developing software for customers insures the enduring success of a company. In continuous development, however, software complexity tends to increase gradually, the consequence of which is deteriorating maintainability over time. During short periods of time, the gradual complexity increase is insignificant, but over longer periods of time, complexity can develop to an unconceivable extent, such that maintenance is no longer profitable. Thus, proactive complexity assessment methods are required to prevent the gradual growth of complexity and instead build quality into developed software.

Many studies have been conducted to delineate methods for complexity assessment. These focus on three main areas: 1) the landscape of complexity, i.e., the source of the complexity; 2) the possibilities for complexity assessment, i.e., how complexity can be measured and whether the results of assessment reflects reality; and 3) the practicality of using complexity assessment methods, i.e., the successful integration and use of assessment methods in continuous software development.

Partial successes were achieved in all three areas. Firstly, it is clear that complexity is understood in terms of its consequences, such as spent time or resources, rather than in terms of its structure per se, such as software characteristics. Consequently, current complexity measures only assess isolated aspects of complexity and fail to capture its entirety. Finally, it is also clear that existing complexity assessment methods are used for isolated activities (e.g., defect and maintainability predictions) and not for integrated decision support (e.g., continuous maintainability enhancement and defect prevention).

This thesis presents 14 new findings across these three areas. The key findings are that: 1) Complexity increases maintenance time multifold when software size is constant. This consequential effect is mostly due to a few software characteristics, and whilst other software characteristics are essential for software development, they have an insignificant effect on complexity growth; 2) Two methods are proposed for complexity assessment. The first is for source code, which represents a combination of existing complexity measures to indicate deteriorating areas of code. The second is for textual requirements, which represents new complexity measures that can detect the inflow of poorly specified requirements; 3) Both methods were developed based on two critical factors: (i) the accuracy of assessment, and (ii) the simplicity of interpretation. The methods were integrated into practitioners' working environments to allow proactive complexity assessment, and prevent defects and deteriorating maintainability.

In addition, several additional key observations were made: Primarily the focus should be in creating more sophisticated software complexity measures based on empirical data indicative of the code characteristics that most influence com-

plexity. It is desirable to integrate such complexity assessment measures into the practitioners' working environments to ensure that complexity is assessed and managed proactively. This would allow quality to be built into the product rather than having to conduct separate, post-release refactoring activities.

Keywords: complexity, metric, measure, code, requirement, software quality, technical risk, technical debt, continuous integration, agile development

ACKNOWLEDGEMENTS

This thesis is the culmination of five years of research that I have carried out at the University of Gothenburg and collaborating companies. I have worked with many professionals who have profoundly influenced me. Their traces can be found throughout this work.

I express my deep gratitude to my main advisor, Miroslaw Staron, my second advisor, Anna Sandberg, and my examiner, Jörgen Hansson, for their invaluable advice and support throughout these years. Their care and professionalism underpin the success of this thesis.

This research was conducted in "Software Center", a research consortium of universities and companies that aims to enhance software engineering practices in industry. I thank the Head of Software Center, Jan Bosch, and collaborators from the companies who enriched my professional life: Wilhelm Meding, Per Österström, Micael Caiman, Johan Andersson, Jesper Derehag, Erik Wikström and Henric Bergenwall from Ericsson; Anders Henriksson, Johan Wranker, Mattias Runsten, and Andreas Longard from Volvo Group Truck Technology; Kent Niesel, Carina Fransson, Jan-Åke Johnson, Darko Durisic and Lars Ljungberg from Volvo Car Group; Christoffer Höglund, Jonas Lindgren and Per Wall from Saab; Laith Said and Gert Frost from Grundfos; Ali Shahrokni from Systemite.

I thank my three friends at work, Alessia Knauss, Lucas Gren, and Siranush Kosayan, who curiously yet unintentionally influenced my work with their advice. Thanks also to all of my colleagues and fellow Ph.D. students for their pleasant presence in my professional life – I apologize for not mentioning their names, but inadvertently omitting anyone would be unfair. Thanks to my parents and friends in Armenia who supported me without questioning the feasibility of my goal. Thanks to Ulrika Kretz for her unconditional support throughout this journey. Finally, thanks to Jack Riganyan who, 9 years ago in the army, having heard my worries of not pursuing our aims told me, "You are right – it's too challenging. But then, you know, we are the challenge lovers."

INCLUDED PUBLICATIONS

This thesis is based on the following studies.

1. Vard Antinyan, Miroslaw Staron, Wilhelm Meding, Per Österström, Anders Henriksson Jörgen Hansson, "Monitoring evolution of code complexity and magnitude of changes." Published in Journal of Acta Cybernetica (0324-721X, Vol. 21, pp. 367-382), 2014.
2. Vard Antinyan, Miroslaw Staron, Wilhelm Meding, Per Österström, Erik Wikström, Johan Wranker, Anders Henriksson, Jörgen Hansson, "Identifying risky areas of software code in Agile/Lean software development: an industrial experience report". Published in 21th IEEE International Conference on of Software Analysis, Evolution and Reengineering (SANER, former CSMR and WCRE conferences), pp. 154-163, IEEE 2014.
3. Vard Antinyan and Miroslaw Staron, "Rendex: A method for automated reviews of textual requirements". Published in Journal of Systems and Software. DOI /10.1016/j.jss.2017.05.079. Elsevier, 2017.
4. Vard Antinyan, Miroslaw Staron, Anna Sandberg, "Evaluating code complexity triggers, use of complexity measures, and the influence of code complexity on maintenance time". Published in Empirical Software Engineering Journal. DOI: 10.1007/s10664-017-9508-2. Springer, 2017.
5. Vard Antinyan, Jesper Derehag, Anna Sandberg, Miroslaw Staron, "Mythical unit test coverage". Published in IEEE Software Magazine, 2017 (scheduled for printing).
6. Vard Antinyan, Miroslaw Staron, Anna Sandberg, & Jörgen Hansson "Validating software measures using action research a method and industrial experiences". Published in 20th International Conference on Evaluation and Assessment in Software Engineering (EASE), p. 23. ACM, 2016.

OTHER PUBLICATIONS

1. Vard Antinyan, Anna Sandberg, and Miroslaw Staron, “A pragmatic view on code complexity management”. Under revision in IEEE Computer Magazine. IEEE, 2017.
2. Vard Antinyan and Spyridon Maniotis. “Monitoring risks in large software development programs”. Published in Computing Conference. IEEE, 2017.
3. Vard Antinyan and Miroslaw Staron. “Proactive reviews of textual requirements.” Published in 24th International Conference on Software Analysis, Evolution and Reengineering (SANER pp. 541-545). IEEE, 2017.
4. Lucas Gren and Vard Antinyan. “On the relationship between unit testing and software quality”. Published in Conference on Software Engineering and Advanced Applications (SEAA), pp. 52-56. IEEE 2017.
5. Vard Antinyan and Miroslaw Staron, “A complexity measure for textual requirement”. Published in International Conference on Software Process and Product Measurement (IWSM-MENSURA 2016, pp. 66-71). IEEE, 2016.
6. Anna Börjesson Sandberg, Miroslaw Staron, Vard Antinyan: “Towards proactive management of technical debt by software metrics”. Published in 15th Symposium on Programming Languages and Software Tools (SPLST’15 pp. 1-15). 2015.
7. Vard Antinyan, Miroslaw Staron, Jesper Derehag, Mattias Runsten, Erik Wikström, Wilhelm Meding, & Jörgen Hansson. “Identifying complex functions: by investigating various aspects of code complexity”. Published in Science and Information Conference (SAI 2015, pp. 879-888). IEEE, 2015.
8. Vard Antinyan, Miroslaw Staron, Wilhelm Meding, Anders Henriksson, Jörgen Hansson, & Anna Sandberg, “Defining technical risks in software development”. Published in International Conference on Software Process and Product Measurement (IWSM-MENSURA 2014 pp. 167-182). IEEE, 2014.
9. Vard Antinyan, Miroslaw Staron and Wilhelm Meding, “Profiling pre-release software product and organizational performance”. Software Center, book chapter (pp. 167-182). Springer, 2014.
10. Vard Antinyan, Miroslaw Staron, Wilhelm Meding, Per Österström, Anders Henriksson, Jörgen Hansson, "Monitoring evolution of code complexity in Agile/Lean software development: A case study at two companies". Published in 13th Symposium on Programming Languages and Software Tools (p. 1-15), 2013.
11. Vard Antinyan, Anna Sandberg, and Miroslaw Staron, “Code complexity assessment for practitioners”. Submitted to International Conference on Software Engineering, 2017.

TABLE OF CONTENTS

Abstract ..... **v**
Acknowledgements ..... **vii**
Included Publications ..... **ix**
Other Publications ..... **x**
Introduction to Software Complexity 17
1 Introduction 18
1.1 The Challenge of Software Complexity 18
1.2 Complexity Assessment 19
1.3 The Need for Proactive Assessment 20
1.4 The Overarching Research Question 21
2 Theoretical Framework 21
2.1 Software Complexity 21
2.1.1 Conceptualization 22
2.1.2 Definition 23
2.2 Software Complexity Assessment 24
2.2.1 Measurement 24
2.2.2 Software Complexity Measures 26
2.2.3 Measurement Validity 27
2.3 Continuous Software Development 29
3 Research Methodology 30
3.1 Action Research 33
3.2 Survey 35
3.3 Case Study 36
4 Research Questions and Contributions 37
5 Discussion 40
5.1 Software Complexity Assessment 40
5.2 Proactive Complexity Assessment in Continuous Development 44
5.3 Software Complexity Landscape 45
6 Limitations 46
7 Further Work 46
Monitoring Evolution of Code Complexity and Magnitude of Changes 49
Abstract 50
1 Introduction 51
2 Related Work 51
3 Design of the Study 53
3.1 Studied Organizations 53
3.2 Units of Analysis 53
3.3 Reference Group 54
3.4 Measures in the Study 54
3.5 Research Method 55
4 Analysis and Results 55
4.1 Evolution of the Studied Measures over Time 55
4.2 Correlation Analyses 58
4.3 Design of the Measurement System 60

5 Threats to Validity ... 62
6 Conclusions ... 63
Identifying Risky Areas of Source Code in Agile Software Development ... 65
Abstract ... 66
1 Introduction ... 67
2 Agile Software Development ... 68
3 Study Design ... 68
3.1 Industrial Context ... 68
3.2 Reference Groups at the Companies ... 69
3.3 Flexible Research Design ... 69
3.4 Definition of Measures ... 71
4 Results ... 73
4.1 Correlation Analysis ... 73
4.2 Selecting Measures ... 76
4.3 Evaluation with Designers and Refinement of the Method ... 76
5 Evaluation ... 79
5.1 Correlation with Error Reports ... 79
5.2 Evaluation with Designers in Ongoing Projects ... 79
5.3 Impact on Companies ... 80
6 Related Work ... 81
7 Threats to Validity ... 82
8 Conclusions ... 83
A Method for Automated Reviews of Textual Requirements ... 87
Abstract ... 88
1 Introduction ... 89
2 Collaborating Software Organizations and Their Requirements ... 90
3 Internal Quality Measurement Model of Requirements ... 92
4 Defining the Measures ... 94
4.1 The Number of Conjunctions as a Complexity Measure (NC) ... 95
4.2 The Number of Vague Phrases as a Complexity Measure (NV) ... 96
4.3 The Number of References as a Coupling Measure (NR) ... 97
4.4 The Number of References to External Documents as a Coupling Measure (NRD) ... 98
4.5 The Number of Words as a Size Measure (NW) ... 99
4.6 Measures Considered but Not Used ... 99
4.7 Range of Measurement Values ... 100
5 Research Design ... 101
5.1 Action Research for Designing Measures ... 101
5.1.1 Access to the data ... 102
5.1.1 Design measures ... 102
5.1.2 Apply measures ... 102
5.1.3 Evaluate measures ... 102
5.2 Developing Rendex ... 103
5.3 Evaluating the Ranking Accuracy of Rendex ... 104
5.3.1 The first approach: evaluating Q_{IR} against Q_{IE} ... 104
5.3.2 The second approach: regression analyses for obtaining Q_{IR} ... 106

5.3.3 Establishing the evaluation setup in the companies 106
5.4 Evaluating Rendex in Companies 107
6 Results of Correlation Analyses and Selection of Measures 107
7 Combining Selected Measures 109
8 Evaluation Results of Rendex 109
8.1 Results of Evaluating QI_R against QI_E 110
8.2 Results of Regression Analyses 111
8.3 Generalizing the Results 112
9 Requirements Quality Index Applied in the Companies 113
10 Threats to Validity 114
11 Related Work 116
12 Summary 118
Evaluating Code Complexity Triggers 121
Abstract 122
1 Introduction 123
2 The Landscape of Code Complexity Sources 125
3 Research Design 127
3.1 Demographics and the Related Questions 128
3.2 Selected Code Characteristics as Complexity Triggers 129
3.3 Complexity and Internal Code Quality Attributes 131
3.4 Selected Complexity Measures 132
3.5 Complexity and Maintenance Time 133
3.6 Data Analysis Methods 134
3.6.1 Evaluating the Association between Job Type and Assessment of Code Characteristics 135
3.6.2 Evaluating the Association between Experience and Assessment of Code Characteristics 136
3.6.3 Evaluating the Association between Type of Job and Assessment of Complexity Influence on Maintenance Time 137
3.6.4 Evaluating the Association between Experience and Assessment of Complexity Influence on Maintenance Time 138
4 Results and Interpretations 138
4.1 Summary of Demographics 138
4.2 Code Characteristics as Complexity Triggers 140
4.3 The Influence of Complexity on Internal Code Quality Attributes 142
4.4 The Use of Complexity Measures 143
4.5 Influence of Complexity on Maintenance Time 146
4.6 Cross-Sectional Data Analysis Results 147
4.6.1 Type of job and assessment of code characteristics 147
4.6.2 Experience and code characteristics 148
4.6.3 Type of job and complexity influence on maintenance time 149
4.6.4 Experience and complexity influence on maintenance time 149
5 Discussion 149
6 Validity Threats 151
7 Related Work 153
8 Conclusions 154

Mythical Unit Test Coverage 157
Abstract 158
1 Test Coverage Measures 159
2 Existing Studies 159
3 The Investigated Product 161
4 Method of Investigation 161
5 Results 162
6 Effect of Complexity on the Results 164
7 Concluding Remarks 166
Validating Software Measures Using Action Research 169
Abstract 170
1 Introduction 171
2 A Recap of Measurement Validation Research in Software Engineering 172
2.1 Theoretical Validation 173
2.2 Validation Using Statistical Models 175
3 A Method for Validating Software Measures 176
4 An Illustrative Case 179
5 Organizational Context of This Experience Report 181
6 Results from Validating Measures in Companies 182
6.1 Measures of Source Code 182
6.1.1 Size 182
6.1.2 Complexity 183
6.1.3 Evolution 183
6.1.4 Defects 184
6.2 Measures of Simulink Models 185
6.3 Measures of Textual Requirements 186
6.4 Summary of Measures and Validation 187
7 Discussion 187
8 Conclusions 188
References 191

Introduction to Software Complexity

1 INTRODUCTION

The success of software development is determined by such parameters as development cost, product quality, delivery time, and customer satisfaction. Software complexity is widely considered to have a crucial impact on these parameters. There are numerous reports on this subject, two of which is drastically summarized here: First, Charette [1] reports a project failure that cost 600U$ million due to excessively complex software. Furthermore, he indicates that large and complex software projects fail three to five times more often than smaller ones. Glass [2], meanwhile, reported that in practice, there is a hundred-percent increase in the software solution's complexity for every ten-percent increase in problem complexity.

Software complexity is influenced by such factors as the product size, product maturity, problem domain, programming languages, development methodologies, and the knowledge and experience of developers. For example, linearly increasing software size is considered to trigger an exponential increase in its complexity [3] and that excellent programmers can be thirty times better in complexity management than average programmers [2].

Nevertheless, a still challenging task is to determine the exact source of software complexity and how it can be proactively assessed for successful management.

1.1 The Challenge of Software Complexity

Software is structurally sophisticated, representationally abstract and progressively versatile over time [4], [5]. These *structural*, *representational* and *evolutional* aspects have a strong impact on software development.

Structural Aspects. Software consists of many elementary units, such as operators, variables, function calls, branching statements, looping statements, pointers, preprocessors, etc. Thousands of *interactions* of these *elements* are the source of the convoluted structure of software. Moreover, multiple artifacts exist in large software development products, such as the requirements and tests necessary for software development. These artifacts augment the challenge of software elements and interactions.

Representational Aspects. Software is abstract; it cannot be touched, felt or observed geometrically like other human-made artifacts. The primary interaction with software is via the computer screen. Software is the only human-made artifact constructed with the help of representational languages, also known as programming languages. The latter are similar to natural languages, one fundamental difference being that making errors in programming languages can have severe consequences. Over the past decades, programming language designers have strived to create as simple and clear languages as possible so that descriptions of machine instructions are straightforward to understand and communi-

cate among software engineers. Language-based representation, however, is still the dominant method of reading and evaluating software.

Evolutional Aspects. Software is progressively versatile – almost any software being used is under active maintenance. Maintenance activities change the representational and structural conditions of software, i.e., while maintaining software, software representation and structure are partly changed. Thus, during the maintenance time, practitioners must understand the current state of the software in order to progress maintenance yet another step.

These three challenges introduce substantial difficulty to software development. Software practitioners refer to these challenges as *software complexity*. In this thesis, we have defined software complexity as:

> *"An emergent property of structural, representational and evolutional aspects of software elements and interconnections that influences software understanding".*

This high-level definition of complexity is based on that of Rechtin and Maier [6] in software architecting, the foundations of which will be discussed in detail later.

Software complexity is highly associated with system understandability. Although complexity is not a thoroughly defined concept, it is still widely used to describe the difficulty of system understanding due to the sophisticated relationship between software elements. Increasing complexity indicates decreasing understandability of software. Therefore, it is natural that complexity both decelerates the development speed and decreases software maintainability and quality [7].

Several definitions of software complexity have been proposed previously; all, however, depend upon the consequences of complexity rather than its essence. For example, Basili [8] defines software complexity as a measure of the resources allocated by a system or human while interacting with a piece of software to perform a given task. Similarly, Zuse [9] describes software complexity as the difficulty in understanding, changing, and maintaining code. Nonetheless, it is vital to understand software complexity in the context of human-software interaction so that software complexity, as perceived by humans, can be measured and managed. Consequently, it is important to scrutinize the source of software complexity and how it affects the work of software practitioners.

1.2 Complexity Assessment

Thomas McCabe and Maurice Halstead were among the pioneers of software complexity measurement who introduced the first measures [10]. Other measures were subsequently introduced [11], such as the information flow measures proposed by Henry and Kafura [12] and measures of object-oriented design proposed by Chidamber and Kemerer [13].

Notably, of all software-related attributes (product, process, project), complexity is the most frequently measured attribute (19% of the time) [14]. How good or appropriate existing measures are, however, has been debated by researchers and practitioners because complexity is not an attribute simple enough to be measured with one measure.

To overcome the difficulty in understanding how good or appropriate a complexity measure is, several studies were conducted to formalize the prerequisite properties of a complexity measure. Then, based on these properties, theoretical validation frameworks were introduced [15], [16]. As expected, in practice, theoretical validation was found to be unsatisfactory in classifying a measure good or appropriate for practical application. Hence, empirical validation emerged as being essential.

Empirical validation is a prerequisite for understanding how effectively a complexity measure indicates problem areas of a piece of software. This was investigated in a number of studies that focused on different forms of correlational or regression analyses, where the relationship of complexity and different kinds of software problems were evaluated. The most common type of analyses investigated the relationship between complexity and defects, resulting in defect prediction models. Despite considerable success in this line of research, practitioners seemed to need clearer guidance on the use of these defect prediction models. Existing complexity measures can be used to predict defects like the problems' symptoms, but not to understand and eliminate the problems per se. Isolated defect predictions turned out to be an insignificant support for practitioners [17]. In practice, practitioners need measurement-based methods that will indicate problem areas, reveal the essence of the problems, and guide problem solving. Hence, it is important to develop complexity assessment methods that indicate problem areas simply and directly and aid practitioners' decision-making for improvement.

1.3 The Need for Proactive Assessment

Continuous software development relies on incremental requirement specification, design, testing and software integration [18]. One of the challenges of continuous software development is to shorten the feedback loops on software artifacts. Shortened feedback loops allow practitioners to track and solve emerging problems more quickly than they escalate into problems with multiple magnitudes of increased cost [19]. When feedback is instantaneous, i.e., "just in time" of problem creation, practitioners can manage the problems proactively to prevent problem escalation. If complexity is assessed "just in time" of development, practitioners can prevent complexity from increasing, thereby reducing the risk of defects and degrading maintainability. Ultimately, this will increase the product quality and reduce maintenance costs.

1.4 The Overarching Research Question

Sections 1.1–1.3 described three research gaps that fundamental to the three main areas of research in this thesis. Specifically these are to:

1. Scrutinize the source of software complexity and how it affects software practitioners' work
2. Develop complexity assessment methods that indicate problem areas simply and accurately
3. Investigate methods for proactive software complexity assessment in practice.

These areas of research are encapsulated in the following research question:

> *How can we proactively assess software complexity in continuous software development?*

The three areas, *software complexity landscape*, *software complexity assessment,* and *proactivity of assessment* in *continuous software development,* are shown in Figure 1.

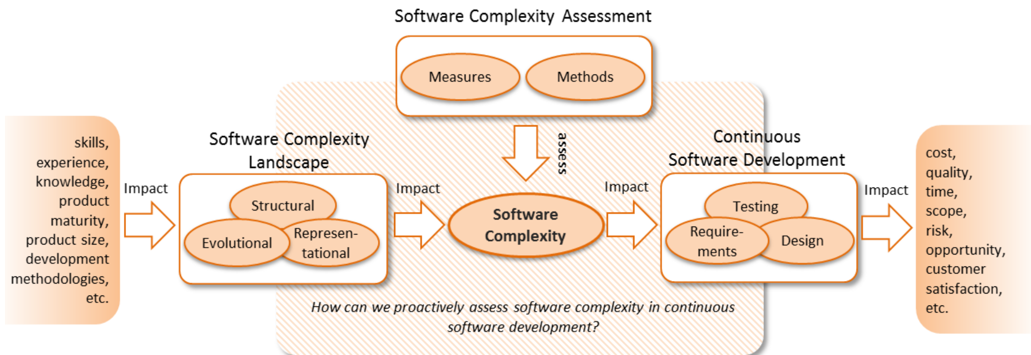


Figure 1 Research focus of this thesis

2 Theoretical Framework

Section 2 describes the concept, history, and modern view of complexity (Section 2.1), the theoretical basis for assessing complexity (Section 2.2), and the influence of complexity on continuous software development (Section 2.3).

2.1 Software Complexity

To understand the source of complexity, the landscape of complexity is explored in this section: first the landscape of complexity generally, and then, that of software complexity particularly.

2.1.1 Conceptualization

In order to understand the essence of complexity, we explore the historical knowledge on complexity. This knowledge has emerged as an epistemological part of the term complexity and is axiomatic by its nature. Edmonds [20] meticulously discussed this knowledge, which can be summarized in four points:

1. Complexity is a property of a system that emerges from the main substance comprising the system, i.e., system elements and interconnections
2. The complexity of a system is only relevant through interaction with another system (typically with humans)
3. Complexity can only be ascribed to a system if the latter can be represented in terms of a communicable language
4. System evolution triggers complexity evolution over time.

The first point suggests that complexity emerges from elements and interconnections, i.e., substances that the system is made of, and also emphasizes the fact that complexity is an intrinsic property of a system.

The second point implies that the complexity of any system either does not exist or is irrelevant if there is no observer. Simply stating, complexity only makes sense when observed from a certain standpoint (typically by a human). A human interacts with a system and acquires information about different elements and their interconnections to understand how the system operates; the notion of complexity emerges through this interaction and compulsion to understand.

The third point suggests that the complexity of a system can only be experienced via a language through which the system is communicated. Therefore, we must distinguish two aspects of system complexity – *structural* and *representational*. The structural aspect requires an understanding of the actual system elements and their interconnections. The representational aspect requires an understanding of the language describing these elements and their interconnections. It is natural to assume that humans cannot skip the representational aspect and directly try to understand the structural aspect because the system must be represented in some sort of language. In the case of software systems, the languages of representation are usually programming languages.

Finally, the fourth point suggests that there is also an evolutionary aspect to complexity in continuously developing systems. This is the complexity caused by the constant change of system elements and interconnections. *Structural* and *representational* complexities do not change in static systems. In evolving systems, however, information about the system elements and interconnections continuously changes and new information is being constructed; a human must learn the new information in order to understand the system operations. A faster-evolving system will generate more new knowledge, thus requiring more effort for new knowledge appropriation. This is the *evolutional* aspect of complexity.

We consider any software system as a typical dynamic system with evolving elements and interconnections. Therefore, we consider the aforementioned factors relevant for software systems.

2.1.2 Definition

According to the IEEE standard computer dictionary, software complexity is defined as "the degree to which a system or component has a design or implementation that is difficult to understand and verify" [21]. According to Zuse [9], the true meaning of code complexity is the difficulty to understand, change and maintain code. Fenton and Bieman [22] view code complexity as the resources spent on developing (or maintaining) a solution for a given task. Similarly, Basili [8] views code complexity as a measure of the resources allocated by a system or human while interacting with a piece of software to perform a given task. Although these definitions recognize the fact that the difficulty of understanding stems from complexity, they do not explore the composition of complexity.

Briand, et al. [23] suggest that complexity should be defined as *an intrinsic attribute* of software as opposed to its perceived difficulty, whilst in information theory, Kolmogorov [24] defines complexity as *the minimum possible length of a system description in some language*. It is not straightforward to calculate the minimum possible length of a system; however, the elegance of this definition is its focus on the essence of complexity and its measurement. It directly indicates that the minimum possible length of a system description can be a measure of complexity.

In software architecting, Rechtin and Maier [6], as well as Moses [25], define complexity as *an emergent property of a system due to interconnections of system elements*. This definition provides the main substance from which complexity emerges—*elements and interconnections*. Based on the discussion in section 2.1.1 and supported by the definition of complexity of Rechtin and Maier [6], we describe the concept of software complexity according to the following five points:

1. *Complexity is an emergent property of software due to software elements and interconnections*
2. *Complexity increases with increasing number and variety of elements and interconnections*
3. *Complexity is experienced through the language through which the software is represented*
4. *Complexity has at least three distinct aspects: structural, representational and evolutional*
5. *Complexity imposes difficulty on humans in software understanding.*

Software systems are developed with programming languages based on accurately defined rules. So how is complexity revealed in programming languages? According to Brooks [4], software complexity emerges from elements and interconnections, such as variables, operators, control statements, preprocessors,

function invocations, etc. Programs containing a greater variety of such elements with denser interconnections are perceived to be more complex. Furthermore, every type of element and interconnection has a different magnitude of influence on complexity.

Mens [5] extends this understanding of complexity by indicating that software elements and interconnections vary in different abstraction levels of the system, such as modules, components and subsystems. Creating different abstraction levels is vital to completely understand the system, although every abstraction level creates its own complexity.

Along with *source code,* which is the core constituent of software products, *requirements specification* and *software tests* are also essential artifacts of software. Requirements and tests can also be described by their complexity. As regards requirements, complexity occurs either in the natural language text or in the models of a system description. Natural language texts and models are alternative descriptions of the software system and, therefore, are equally exposed to complexity. Tests, meanwhile, are similar to code so the complexity is in the code (programming language) used to develop the tests.

The complete picture of software elements and interconnections is still hardly investigated. Moreover, research on the influence of different types of elements and interconnections on complexity is very rare so a part of this thesis is dedicated to this subject.

2.2 Software Complexity Assessment

Section 2 firstly introduces the concept of measurement fundamental to complexity assessment and widely used throughout this thesis. Examples of known complexity measures are then brought, which are used in the current complexity assessment methods. Finally, the need for new methods for advanced complexity assessment is highlighted.

2.2.1 Measurement

Several definitions of measurement exist in the literature. In software engineering, Fenton and Bieman [22] define measurement as the:

> *"Process by which numbers or symbols are assigned to attributes of entities in the real word in such a way as to describe them according to clearly defined rules".*

This definition implies quantification of the attributes of software artifacts, processes or products with clearly defined rules. The definition does not, however, enforce the meaningfulness of measurement, which plays an important role in making observations. Hubbard [26] defines measurement in applied economics as:

> *"A quantitatively expressed reduction of uncertainty based on observations".*

This definition adds a pragmatic value to measurement and is used, therefore, throughout this thesis in conjunction with Fenton's definition. Hubbard's definition implies that any quantification of an *attribute* cannot be called measurement unless that quantification reduces the uncertainty on the *measurement entity*. To understand this statement, one can consider counting the number of methods in two Java programs for comparing program sizes. Since the size of every single method can vary greatly (in terms of *lines of code*), it cannot be concluded from the end result as to which program was larger. In this context, therefore, counting the number of methods is not a measurement. This is crucial from a pragmatic standpoint because any measurement implicitly implies decision support for practitioners.

Figure 2 provides an understanding of measurement as used in our work based on an example for software complexity measurement and distinguishes two worlds—*comparative* and *operationalized.*

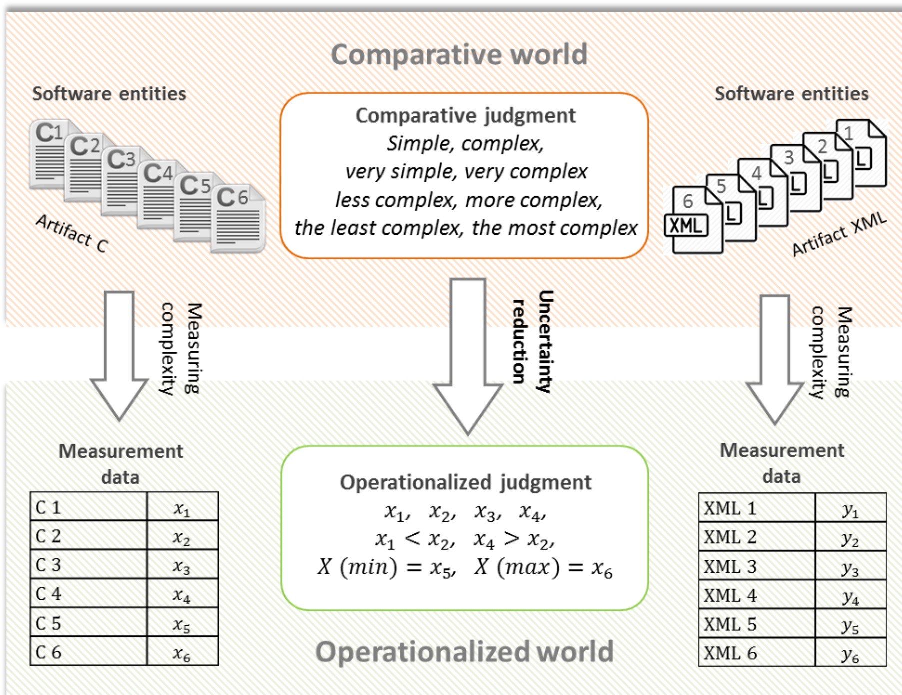


Figure 2 Overview of software measurement

The comparative world represents that in which we compare the complexities of entities based on comparative adjectives of natural language. For example, we can say that *entity C1* is more complex than *entity C2*, or that *entity C6* is the

most complex of all the entities. This kind of comparison is usually based on perceptions. In the operationalized world, the complexity of every entity is a number. Numbers are assigned to the entities according to a predefined rule so that they provide greater precision on the complexity of an entity and thus reduce the initial uncertainty in the comparative judgment. Depending on the software artifact, the rule of assigning numbers can be different. The end results, however, should be artifact-independent numbers that can be subjected to comparison. Figure 2 exemplifies two artifacts: source code files (C) and textual requirements (XML). The comparison of the complexity numbers for artifact C is depicted in the middle part of the operationalized world.

2.2.2 Software Complexity Measures

The first software complexity measures were created in the late 1970s, the most widely-known being the McCabe cyclomatic complexity [10] and Halstead's measures of software science [11]. New and more advanced complexity measures were created subsequently, such as the coupling measures of Henry and Kafura [12] and object-oriented programming measures of Chidamber and Kemerer [13].

The **cyclomatic complexity measure** [10] is based on the control flow structure of a program, calculated as follows:

Cyclomatic number (M) $$M = E - N + 2 \quad (1)$$

M is the cyclomatic complexity number, E is the number of edges, and N is the number of nodes in the control flow graph of the program. An alternative method of calculating M is to count the number of control statements in the program. McCabe created this measure primarily as an aid for software testing. The fact is that with linearly increasing cyclomatic complexity number, the number of execution paths in a program increases exponentially.

The **Halstead [11] measures** are calculated based on the number of operators and operands in a software program. *Operators* are typically all mathematical and logical operators in a program, whilst the *operands* are typically all invocations of variables and functions in a program.

Two of the Halstead measures can be calculated as:

Program volume (V): $$V = N \times log_1(n_1 + n_2) \quad (2)$$

Program volume is meant to estimate the number of bits required to store the abstracted program of length N.

Program difficulty (D): $$D = (n_1/2) \times (N_2/n_2) \quad (3)$$

where:

n_1 *is the number of unique operators*

n_2 *is the number of unique operands*

N_1 *is the number of all operators*

N_2 *is the number of all operands*

$N = N_1 + N_2$

$n = n_1 + n_2$

Program difficulty is meant to estimate the difficulty of the program based upon the most compact implementation of the program. Difficulty increases as the number of unique operators increases.

Henry and Kafura [12] measure is calculated based on *invocations* and *size* of a function (method):

Coupling (C): $$C = LOC \times (fanIn \times fanOut) \quad (4)$$

where:

fanIn is the number of invocations of a given function in a specified program

fanOut is the number of invocations of functions in a given function

LOC is the number of lines of code of a given function

Coupling shows the magnitude of interconnections of a given function/method within a program.

Clearly, the definitions of these measures are based on certain *elements and interconnections* of code. The McCabe complexity is based on *conditional statements*, the Halstead measures are based on *operators* and *operands,* and coupling measures are based on *invocations of functions*. Every measure is designed according to its own rationale as to why certain elements are considered in the complexity measurement and others are not. In the case of cyclomatic complexity, the consequence of control flow was considered because a function with too many decision points is difficult to test. In the case of the Halstead measures, almost all structural elements were used because the program volume and difficulty had to be assessed. In the case of the Henry and Kafura measure, the invocations of functions were used because highly coupled functions are considered to be difficult to maintain.

Notably, each measure assesses a different aspect of software complexity, which appears to have many more aspects and may be the reason why many measures of software complexity are reported in the literature [9]. It may also be the reason why an all-encompassing complexity measure has not yet been created.

2.2.3 Measurement Validity

The validity of complexity measures allows determining how well a measure assesses complexity. Two main clusters of validation methods exist:

1. Theoretical validation, and
2. Empirical validation.

Theoretical validation is based on theoretical validation frameworks, which typically define the prerequisite properties of a complexity measure. A complexity measure is regarded as valid if it possesses these prerequisite properties. Properties are defined based on the cumulative general knowledge on complexity. Notable examples of validation frameworks are provided by Weyuker [15] and Briand, et al. [16].

To elucidate the essence of properties, the properties of complexity proposed by Briand, et al. [16]l can be considered. The authors define the concept of *system* as a representation of system *elements* and their *connections*, such that complexity is defined as a function of the system with the following properties:

1. *Non-negativity:* the complexity of a system is non-negative
2. *Null value:* the complexity of a system is 0 if the relations of elements are non-existent
3. *Symmetry:* the complexity of a system does not depend on the convention chosen to represent the relations between its elements.
4. *Module monotonicity:* the complexity of a system is not less than the sum of the complexities of any two of its modules with no relationships in common
5. *Disjoint module additivity:* the complexity of a system composed of two disjoint modules is equal to the sum of complexities of the two modules.

These properties are defined in order to facilitate the design of complexity measures. Notably, several frameworks in the literature define properties for complexity. Naturally, the different frameworks propose different properties of complexity because they envision different motivations behind the properties. Complexity, however, is not a well-defined concept, even in older and more mature fields of science. Therefore, when considering pragmatic tasks, such as complexity measurement, it has been difficult to define the prerequisite properties of a complexity measure. For example, the third property of complexity according to Briand, et al. [16] implies that the language of software representation does not influence complexity measurement, whereas in practice, language-dependent features, such as deep nesting or misplaced indentations, can be perceived as manifestation of complexity.

Empirical validation is based on the assessment of the predictive power of the measures. Most of the time, the complexity measurement *per se* is not of ultimate interest for practitioners. Rather, it is used to predict the extent to which complexity impacts business factors, such as quality, risks, time, cost, effort and developers' work. Empirical validation suggests that complexity measures must be good predictors of such factors [27]. Thus far, however, defect prediction [28] has primarily been used for empirical validation of complexity measures. The number of defects has been seen as a substitute of software quality. The most likely reason for the popularity of defect prediction is that measuring the number of defects has been relatively easier than measuring effort and cost.

Despite the advances in complexity measurement and validation, serious issues must still be addressed. For example, Fenton [29] highlights the commonly held viewpoint that a complexity measure is not valid unless it is a good predictor of a particular attribute. A consequence of this is that pure size measures have been regarded as useful measures as they are good predictors of defects, whereas deeper scrutiny shows that the predictive power of size measures is based purely on probabilistic reasons. Naturally, larger programs have more defects. This prediction, however, is not useful for quality improvement purposes so in practice, these prediction models are not used to help practitioners develop better code [30]. Methodological problems of measurement validity, highlighted by Kitchenham [14], emerge from following a validation methodology without first reflecting on its adequacy.

2.3 Continuous Software Development

Continuous software development is defined as a:

> *"Software engineering approach in which teams produce software in short cycles, ensuring that the software can be reliably released at any time. It aims at building, testing, and releasing software faster and more frequently" [31].*

Software development companies transition towards continuous software development because it facilitates waste reduction in the development chain [32], [33]. Other benefits include reduced deployment risk, easier assessment of development progress and shorter loops of user feedback [34].

Challenges also exist. All the development processes are carried on in a continuous manner: continuous planning, coding, integration, testing, deployment, and maintenance. In this environment, developers want faster feedback on newly delivered software yet comprehensive reviews and tests take a long time to run. The risk then is the gradual degradation of software maintainability and late design modifications.

Humble and Farley [34] have discussed five pivotal activities that underlie continuous development. In order to succeed in continuous software development, software developers should:

1. Build quality into the product in the first place
2. Work on small batches by getting every change in version control as far towards release as possible
3. Solve the problems, leaving repetitive tasks to the computers
4. Relentlessly pursue continuous improvement
5. Be responsible for the quality and stability of the entire software being built.

It is remarkable that all five activities aim to decrease the risk of gradual software degradation. In particular, the first activity (derived from the Deming's Third Principle of Lean Thinking [35]: "…eliminate the need for inspection on a

mass basis by building quality into the product in the first place") intends to mitigate the risk of code degradation. This paradigm shift views quality improvement as an integrated activity rather than a separate inspectional one. Instead of inspecting/testing software after development and reacting to areas of degraded software (revealed by defects and unmaintainable code), the rationale here is that quality is built into the software proactively. Proactive behavior is defined as:

> *"…the relatively stable tendency to effect environmental change" [36].*

Proactivity assumes that software developers are well aware of the quality of their software before its integration so any feedback on their software after integration cannot trigger reactive actions. Proactivity, however, requires integrated methods for providing feedback to the developers "just in time" of development so that they can prevent software degradation.

Since software complexity is one of the major reasons for software degradation, complexity management should also be carried out proactively; developers should be able to obtain feedback on complexity "just in time" of software development, allowing them to take instrumental action immediately.

3 RESEARCH METHODOLOGY

Section 3 describes the rationale for the choice of research methodology. The three main research methods employed in this work are presented.

The research context had three key characteristics of fundamental importance to the choice of research methodology:

1. The research context was highly *sophisticated*, due to the involvement of multiple software development artifacts, processes and human factors in the large development projects.
2. The *scope* of the research problem was *extensive*. The effect of complexity permeated into different software development artifacts and people's professions and so had different manifestations and subsequently different interpretations.
3. The goal of the research was to attain *applicable* results so the research method assumed a reflective nature that would allow feedback from practitioners to calibrate the results.

The sophisticated context, extensive scope and the requirement for results to be applicable limited the option of employing a purely *positivistic* approach [37] in this research. More specifically, the phenomenon of complexity could not be described with a minimal set of general variables across all contexts and for all artifacts. In addition, since complexity was perceived differently by practitioners of different professions, the interpretative nature of the results had to be part of the final solution and a certain degree of *Interpretivism* [37] was required in the research methods used.

On the one hand, complexity could be *measured* within certain boundaries determined by the type of software artifacts, programming languages, organizations, product domains and practitioners' professions. This meant that within certain boundaries, a theory based on deductive reasoning could be postulated and subsequently evaluated in practice (typical to positivistic thinking). On the other hand, the same theory should be subjected to application and generalization across boundaries, meaning that it should be tested across boundaries to allow a wider understanding and gradual theory building (typical to interpretivistic thinking).

The existence of positivistic and interpretivistic elements in the research methodology indicates methodological *realism* (Figure 3, Layer 1). This philosophy is similar to positivism, but recognizes that all observations are fallible to a certain degree and, therefore, all theories are gradually improvable. The aforementioned factors suggest that the research methods of this thesis should be based upon methodological *realism*.

Additionally, the research demanded the *applicability* of research results; this was not straightforward because of the sophisticated research context. More specifically, it was not easy to use a specific method for creating complexity measures for one company and simply applying these measures to other companies. It was not generally possible to crystalize the conditions of the research environment that would facilitate the *repeatability* of research results. Repeatability is the major issue for sophisticated organizations' sciences [38]. To overcome the issue, Checkland and Holwell [38] proposed that this criterion can be replaced by a *recoverability* criterion. The essential idea is that anyone interested in subjecting the research to critical scrutiny can get full access to the research process. Having fully recoverable research allows the sophisticated nature of research context to be understood sufficiently; this can be valuable in designing similar studies (documenting similarities and differences).

A method that embraces methodological realism and is applicable for immediate problem solving is *action research* [39]. Therefore, action research was employed as the main method for scientific inquiry in this thesis, thereby allowing the results to be *applied* in the companies. Action research is perfectly suited to this purpose because of its practical problem solving. We, researchers had the opportunity to work alongside practitioners to acquire valuable qualitative and measurement data, which was advantageous in conducting a typical action research process in the companies.

The second method used in this research was *survey* because the collective viewpoint of practitioners could reveal important facts about software complexity. Previously, theoretical explanations of software complexity have been very much emphasized. Even complexity measures were essentially based on theoretical considerations. In practice, however, software is always associated with more sophisticated aspects than those assessed by theoretical considerations. For example, it is still unclear how the different professions of people, such as testers, programmers, architects and managers, affect the perception of com-

plexity. Hence, knowledge for software complexity understanding and measurement should also be derived from practical observations. One problem here is that knowledge obtained from a survey can be inconclusive as it is derived from practitioners' perceptions rather than factual sources. Nevertheless, since the research context itself is complex, there is little chance to obtain conclusive knowledge directly [38], whilst knowledge obtained from a survey is valuable in identifying likely answers. This knowledge can subsequently be used for triangulation and more reliable theory construction.

The third (last) method used in this research was *case study*. As with action research, case study is also suitable for researching sophisticated systems. A fundamental difference, however, is that in case study the researcher becomes a detached observer. The application of case study in our research was essential when an independent understanding of certain problems in the research context was necessary or interesting from a research perspective.

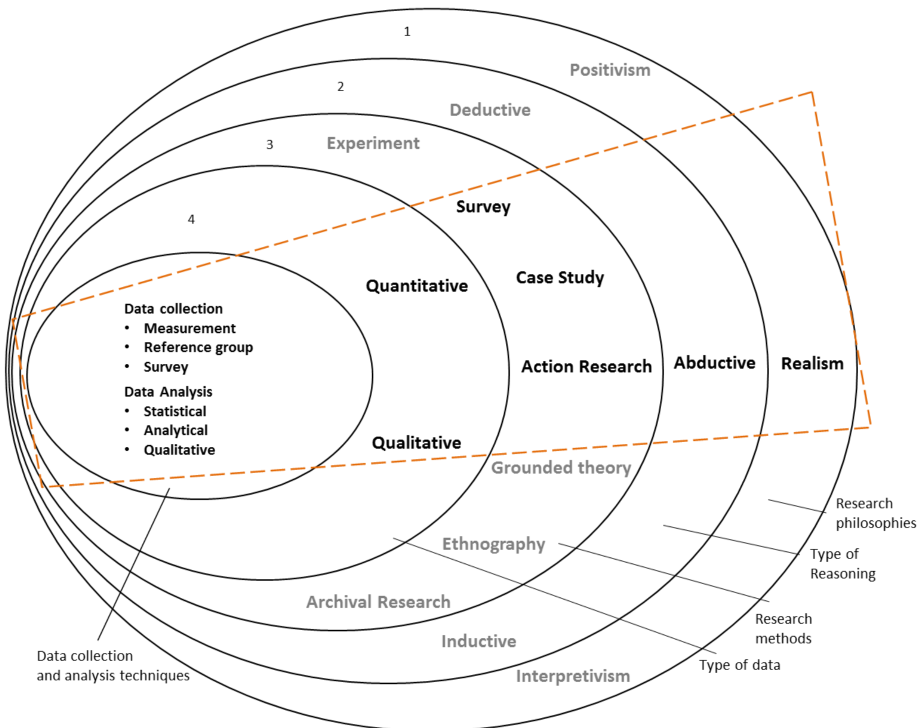


Figure 3 Methodological map of Saunders [40] (pp. 106-135)

An overview of our research methodology based on the methodological map of Saunders [40], the so-called *research onion,* is shown in Figure 3. The first four layers of the research onion encompass four important methodological items: the research philosophy, type of reasoning, research method and type of data. The dashed trapezoid in Figure 3 outlines the boundaries of our research meth-

odology. As explained earlier, the research philosophy relies on methodological *realism* (Layer 1). *Abductive* reasoning characterizes the process of drawing conclusions from the obtained data (Layer 2). Abductive reasoning is typical with case study and action research, where it is not possible to obtain a conclusive data set and, therefore, conclusions are made based on the likeliest possible explanations. *Survey*, *case study* and *action research* methods lie in the upper middle part of the figure (Layer 3). Both *quantitative* and *qualitative* data were collected in our studies (Layer 4). At the core of Figure 3, we have outlined the main techniques of data collection and analysis; these are described in detail in each of the papers comprising this thesis.

The forthcoming sections briefly describe the research methods and their application in this research. Their specific application in each study is described in the corresponding papers of this thesis.

3.1 Action Research

Action research as a research method allows progressive problem solving based on the reflective process of a *researcher-practitioner* collaborative setup. The term *action research* was coined in 1946 by Kurt Lewin [41]. Action research is a cyclical process that includes identifying the client's problem, designing actions, applying the actions to the client's system, finding new facts, reformulating the problem, and repeating the cycle until the problem is meticulously understood and, if possible, solved. The whole process is geared towards incremental problem solving, and, in contrast to other methods, it allows researchers to intervene in the research process by introducing actions.

In the 1970s, organizational science faced a crisis because of the increasing complexity of modern organizations. As reported by Susman and Evered [42] , many of the findings in scholarly journals were (and still are) only remotely related to real-world problems. To overcome the crisis, a group of scientists pioneered the application of action research in organizational science [43], [44], [38]. From the 1990s onwards, when software engineering organizations started to grow rapidly, action research was also applied in these organizations [45], [46].

The companies that participated in this research are typical examples of large, complex organizations. Since the aim of this research included both knowledge acquisition and the solving of practical problems, and because we were able to work closely with practitioners, we found the action research method to be suitable for this work. Consequently, action research was applied to research tasks, such as *developing* and *evaluating*:

1. Complexity measures
2. Complexity assessment methods
3. Measurement systems.

We have applied the typical cycle of the action research process, as prescribed by Susman and Evered [42] (Figure 4). The cycle starts with *diagnosing* either the problem or improvement opportunities in the company. The problem can be either articulated by company representatives and discussed further to gain deeper understanding or identified by researchers in an initial case study. The *action planning* phase includes developing solutions for the problem. Both literature and empirical evidence were used to understand the solution options. The third step in the cycle includes *action taking*, both small-scale actions (e.g., introducing a measure to the organization) and large-scale actions (e.g., introducing an entire measurement system to the organization). The fourth step focuses on *evaluation,* which includes evaluating newly introduced measures, methods, and measurement systems. Step four is done in order to evaluate qualities, such as the efficiency, effectiveness or adequacy of the solutions.

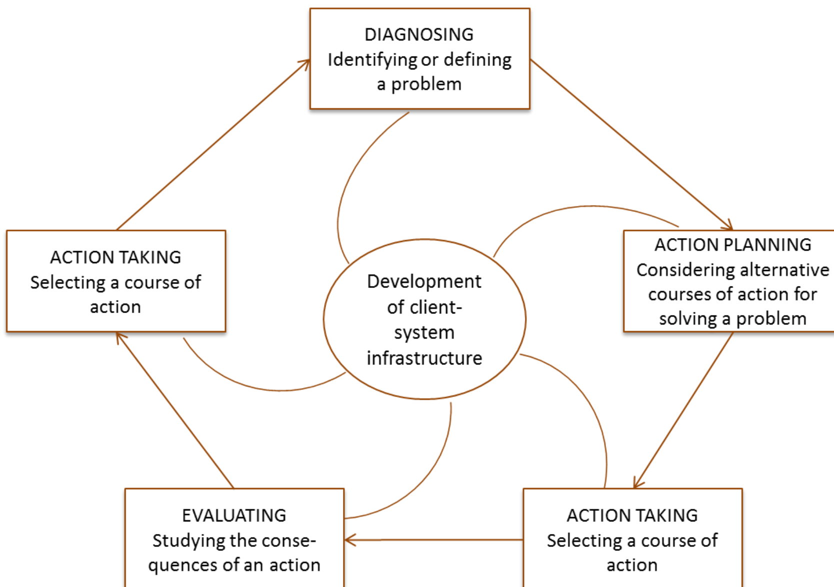


Figure 4 The cyclical process of action research (Susman and Evered [42])

The final step focuses on *learning* from findings and documenting these findings. Here, we reflected on the findings and reformulated the problem for new action planning. The cycle was repeated as often as required to either solve the overall problem or confirm that the adopted approach was unsuitable for problem solving.

The client system infrastructure is shown at the center of Figure 4. The core of the client system infrastructure in our case was a group of practitioners called the reference group, members of which provided evaluative feedback upon the courses of actions.

Reference groups were formed in every company to establish a "researcher-client" collaboration [47], [48]. Reference groups comprised several software engineers and managers who regularly provided feedback on the obtained results. The reference group practitioners were very well-informed on the subject under investigation and usually had deep insights on the results. Their insights helped the interpretation of results, as well as suggestions on how the subject could be investigated further. The professions of reference group practitioners could vary depending on the subjects under investigation. In all cases, however, practitioners were either software developers/testers, software architects or project managers, the latter having excellent knowledge concerning both the technical issues of the products and the managerial issues of decision making.

In addition to the more active involvement of reference group practitioners, we also presented results to other practitioners in the companies, albeit less frequently, so as to gather knowledge from a wider practitioner spectrum. The active involvement of practitioners in the research allowed us to achieve a broader and deeper understanding of the problem domain, obtain results, and assess the potential to apply these results.

3.2 Survey

Survey is a structured means of soliciting information from people [49]. In social science, social psychology, politics and business surveys are widely used to understand peoples' preferences, attitudes and opinions on particular issues. Recently, surveys have been widely used in software engineering to investigate topics like user experience and management practices. The main purpose of conducting a survey is to draw general conclusions based on a fraction of the population. A survey, for example, can be used to explore prospective users' attitudes to a new product feature.

In the research onion of Saunders [40] (Figure 3), the survey more closely resembles the positivistic philosophy because it allows to collect quantitative data and make statistical inferences. Surveys can in fact be used to acquire both quantitative and qualitative data. A guideline for conducting surveys in software engineering has been proposed by Linåker, et al. [50].

In this study, a web-based quantitative survey (Rea and Parker [49] pp. 8-79) was developed in order to reach prospective respondents. Survey was used for three research tasks, i.e., to:

1. Acquire empirical data on software complexity triggers as experienced by software developers
2. Estimate the use of current complexity measures in practice
3. Estimate the impact of complexity on software quality and maintenance time.

The first task was investigated using a survey because research on software complexity understanding and measurement has predominantly been theoreti-

cal so far. Traditionally, research on software complexity has followed positivistic research philosophies, i.e., defining the theoretical properties of complexity (e.g., [15], [16]), developing complexity measures (e.g., [9], [22]), and finally empirically validating these measures against dependent variables, such as defects or maintainability (e.g., [51]). This approach has been only partly successful [17] mainly because, software complexity is more convoluted in practice than in theory. Practitioners' perspectives on software complexity are nearly non-existent in the literature. The use of survey provided a new perspective on complexity understanding and its potential measurement.

The second task was investigated with survey because the use of complexity measures can be directly quantified by asking practitioners whether they use these measures. Asking a large proportion of practitioners (population) this question can provide a realistic estimate of the use of these measures.

The third task was investigated with survey because it is difficult to establish an accurate relationship between maintenance time and complexity. We suggest that the practitioners' collective standpoint on this relationship is a valuable complementary data source for triangulation and more accurate conclusions.

3.3 Case Study

The last and least used method in this research is *case study*. Case study is an empirical enquiry that investigates a contemporary phenomenon in real-life (Yin [52] p. 13). Case studies are suitable for studying a phenomenon where the boundaries between the research context and the phenomenon are not strictly determinable. Above all, when the research phenomenon and context are sophisticated and a more in-depth analysis is preferable, case study is a pragmatic choice. Typically case studies can be adopted for post-facto studies, so the study results do not have any effect on the studied event. It is commonly used in areas, such as psychology, sociology, community planning, etc.

As with action research, case study is effective when investigating complex sociotechnical systems. Yet a significant difference exists between these two methods: the goal of action research is to solve a problem for the client, whilst that of case study is to provide independent and often post-facto analysis. Action research is a strictly iterative process; this implies that researchers will intervene in the system being investigated by introducing changes and studying their consequences. In contrast, case study implies that researchers will study the system with no intervention in the research process ([53] p. 13). Both methods generate knowledge for a particular case, but with different purposes and via different approaches.

In this thesis, we used case study primarily to study the relationship between unit test coverage measures and defect count in one company. This relationship was investigated in the wider context of software size, complexity and evolution. Unlike a typical case study that relies on qualitative analysis, this study used numerous measures to investigate the subject quantitatively and, as it was one

of the two last studies in our research, a wealth of qualitative understanding of the results was obtained based on experiences accumulated from preceding studies. Our other studies in the section of "Other Publications" also contain case studies.

4 RESEARCH QUESTIONS AND CONTRIBUTIONS

Section 4 clarifies the relationship between the research questions and research contributions of this thesis, which are depicted in relation to the three main areas of focus of this research (Figure 5).

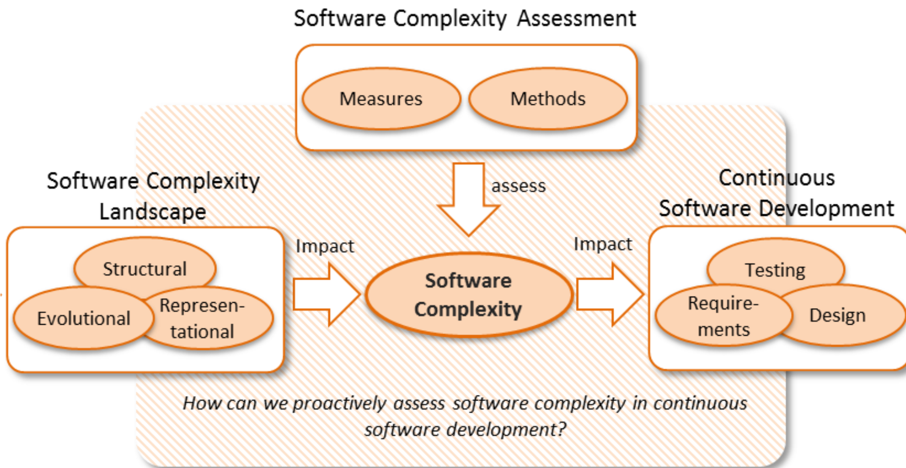


Figure 5. The research focus of this thesis

To recap, the main research question of the thesis is:

> *How can we proactively assess software complexity in continuous software development?*

This research question is based upon the eight, more detailed, original research questions addressed in the six papers comprising this thesis. These eight questions are shown in Table 1.

Research questions 1-3 are related to *proactive complexity assessment* in *continuous software development* (the upper and the right-hand rectangles of Figure 5). In particular, the aim of these questions was to create complexity assessment methods and apply them proactively in order to provide feedback on continuously delivered software.

Research questions 4–6 are related to the *triggers* of complexity, the use of existing complexity *measures* in practice, and the *impact* of complexity on maintenance time (the left-hand rectangle of Figure 5). The aim of these questions was

to fill the practical gaps of understanding complexity, to evaluating the use of the most popular complexity measures in practice, and to estimate the impact of complexity on the software maintenance time as perceived by practitioners.

Finally, research questions 7 and 8 explore the problems and opportunities in the field of software (complexity) *measurement* (upper rectangle of Figure 5). They provide a critique on software complexity assessment methods.

Table 1 Research questions

N	Research Question	Paper
1	How can we monitor code complexity and changes effectively when delivering feature increments to the main code branch?	1
2	How can we identify and assess risky elements of the code effectively when delivering new feature increments to the main code base?	2
3	How can we automatically rank textual requirements based on their internal quality in large software development organizations?	3
4	Which code characteristics are perceived by practitioners as the main triggers of complexity?	4
5	How frequently are complexity measures used in practice?	4
6	How much does complexity affect maintenance time?	4
7	What is the relation between unit test coverage, complexity and defects?	5
8	How can we validate software measures when the variables of prediction are not accurately measurable?	6

Table 2 summarizes our findings after each research question was answered. Findings one, three, five, and six present solutions for *complexity assessment* (the upper rectangle of Figure 5). Findings two, four, and seven present solutions for *assessment proactivity* (the right-hand rectangle of Figure 5). In addition, these first seven findings led to the important conclusions that many current measures of complexity have major shortcomings. Subsequent research, therefore, focused on exploring the triggers of complexity and the possibility of better measurement, which resulted in to findings 8–10 (the left-hand rectangle of Figure 5).

Throughout the entire research process, we also documented valuable experiences that highlighted the challenges and opportunities of *complexity assessment* in practice, as summarized in findings 11–14 (the upper rectangle of Figure 5).

Figure 6 helps to elucidate the relationship between the research questions and subsequent findings. Every research question was answered by one or more findings.

Table 2 Summary of the findings

N	Finding	Paper
1	Only two out of investigated five complexity measures suffice to monitor overly complex functions and files.	1
2	Only a few out of thousands of functions and files increase code complexity of the product when monitoring complexity over a period of weeks.	
3	The product of *cyclomatic complexity* and the *number of revisions* indicates source files that are error-prone and difficult to maintain.	2
4	Incorporating the indicator (Finding 3) into a measurement system allows the proactive identification of error-prone and difficult-to-maintain files.	
5	Four complexity measures have been defined which enable automated complexity measurement of textual requirements.	3
6	The weighted sum of the four measures (Finding 5) indicates requirements that are difficult to understand for implementation and testing.	
7	Incorporating the indicator (Finding 6) within the requirements' management system allows the proactive identification of requirements that are difficult to understand.	
8	Two code complexity triggers (*nesting depth* and *lack of structure*) are considerably more important than other triggers.	4
9	Well-known complexity measures are rarely used in practice.	
10	Practitioners' cumulative perception indicates that complex code consumes multifold more additional maintenance time compared to simple code	
11	Statement, decision and function coverage measures are inadequate for deciding upon the sufficiency of testing	5
12	The maximum level of nesting is the only measure with a tangible effect size on both defects (16%) and coverage (18%).	
13	Empirical validation of measures based on regression analysis is often inaccurate because a dependent variable like defect count often cannot be measured accurately.	6
14	Validation of measures using action research can help to determine the usefulness of the measure in practice.	

The next section provides an overarching discussion of the findings and their implications. Note that we have intentionally avoided providing related background in order to maintain focus. More details on each finding can be found in the corresponding papers.

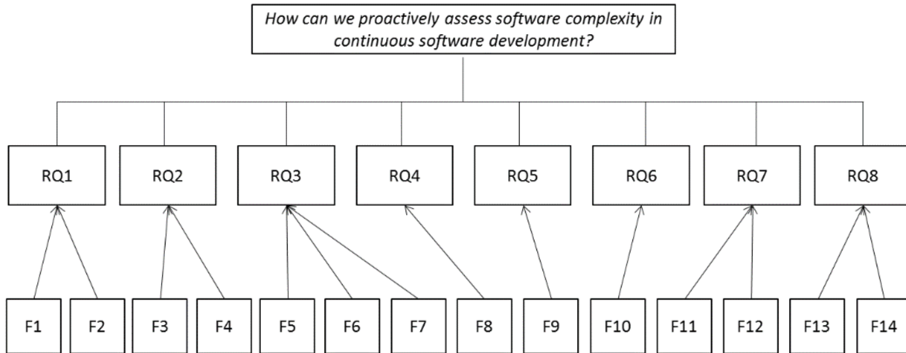


Figure 6 Findings mapped onto the research questions

5 DISCUSSION

In this section, the pivotal points of all 14 findings are discussed within the three, logically separate areas shown in Figure 5: *software complexity assessment*, proactive complexity assessment in *continuous software development*, and *software complexity landscape*.

5.1 Software Complexity Assessment

Finding 1 exemplifies the analysis of the relationship between five code complexity measures. The conclusion that only two out of the five investigated complexity measures are sufficient to monitor complexity in practice reduces the problem of having to use many measures. Two problems predominate:

1. If strongly correlated measures exist, these can indicate the same aspect of complexity [17] so only one of the strongly correlated measures may be useful.
2. It is difficult to use many measures because this makes it harder to interpret the results for final decision making.

Finding 3 is the logical continuation of Finding 1. It represents a means to assess complexity based on the product of the two non-correlated measures: the *effective cyclomatic complexity* (M_{ef}) and *number of revisions (NR)* of a source code file.

$$R = M_{ef} * NR \quad (1)$$

This formula is designed to assess the *relative risk (R)* of error-prone and difficult-to-maintain files. In contrast with existing works, where measures are combined based on regression equations (as demonstrated previously [54]), here the measures have been combined based on their semantic interpretation. The merit of this formula is that it provides interpretable numbers in a practical context. It indicates that if a *structurally complex* file *changes frequently*, there is a risk of errors and degrading maintainability.

Importantly, the formula expresses the fact that if any of the measures obtain a zero value, then the value of risk must also be zero because:

1. If complex files do not change at all ($NR = 0$), then they are not exposed to risk ($R = 0$)
2. If intensively changed files are only simple ones ($M_{ef} = 0$), then they do not indicate risk, but intensive development ($R = 0$).

If we juxtapose formula (1) with defect prediction models based on regression analysis, we have the following disadvantage and advantage, respectively:

1. Formula (1) does not ensure that defects are predicted with the maximum achievable accuracy, while regression equations in the previous works do.
2. Regression equations do not support decision making for code improvements because they have either complicated interpretations or no interpretations at all [17]. Formula (1) does help with decision making because it has clear interpretation based on which instrumental actions can be taken.

In Ericsson, for example, the use of formula (1) could identify files that are frequently changed not because of active maintenance, but because they are simultaneously complex and play a central role in the product code. These files were error-prone and continuously changing over time, inducing new defects, such that the organization decided to modularize them by refactoring.

A problem with this approach is that if any measure in the formula is not a reasonably accurate measure of its purpose, then the whole formula fails. *Cyclomatic complexity* in particular is a moderately good measure of structural complexity. Therefore, the formula should be used to reflect what it shows for a particular case rather than as a decisive indicator. This formula could be greatly improved by using a more sophisticated structural complexity measure, which is a topic for future investigation.

The fifth finding shows that it is possible to define complexity measures for textual requirements. The problem is that natural language is a more sophisticated representational language for a software system than source code. Its meaning lies in whole sentences rather than words as machine instructions in code. Therefore, defining simple measures for a sophisticated representational language is an unlikely possibility [55]. Nevertheless, the research in four com-

panies showed that it was possible to define four simple complexity measures for textual requirements. Three of these measures are completely new and quantify the structural complexity of requirements. The first measure, *number of conjunctions (NC)*, measures the number of interrelated actions in a requirement and is similar to McCabe's cyclomatic complexity [10] in terms of decision sequences. This measure is also the most accurate indicator of internal quality with an effect size on internal quality as great as 59%. Put in perspective of its usefulness, this measure has about as much effect size on internal quality of requirements as popular code complexity measures have on code defects. For example, the results of finding twelve shows that one of the widely used measures, *NR* [56] has 62% effect size and cyclomatic complexity *(M)* has 41% effect size on code defects.

The next two measures are similar to the *fan-out* measure of Henry and Kafura [12]. The *number of reference modules (NRM)* and *number of reference documents (NRD)* are two different measures of structural complexity, which can be classified as measures of coupling according to Briand, et al. [16]. The *NRM* has a best case effect size of 46%. The effect size of the *NRD* was not possible to calculate because of measurement values being too low in magnitude, but it also has substantial effect based on qualitative evaluation. The last measure, the *number of vague phrases (NV)*, is a measure of representational complexity and is based on several previous works, including that of Femer *et al*. [57].

These results presented the opportunity for *automated requirements reviews*, a task regarded as one of the most tiresome activities in software engineering [58]. In contrast with existing work (e.g., [55] and [57]) that primarily focuses on ambiguity measurement, our results provide a means for complexity measurement. The use of both types of measures can lead to the design of better indicators for decision support. To our knowledge, no previous work has evaluated the effect size of ambiguity measures; therefore, we cannot provide any comparison of ambiguity and complexity measures, which would be interesting from the assessment effectiveness perspective.

Finding 6 presents a combination of the four complexity measures into a single formula to assess a requirement's internal quality (*QI*). This combination is based on the simple sum of the measures:

$$QI = NC + NV + NRM + 5 * NRD \qquad (2)$$

As opposed to formula (1), comprised of one structural and one evolutional complexity measure, all four measures in formula (2) are structural measures. Since the evolutional complexity of requirements is low, formula (2) does not contain any evolutional complexity measure.

Finding 11 focuses on three popular unit test coverage measures used to test sufficiency, and indicates that all three measures are inadequate for deciding upon the sufficiency of testing. The effect size of measures on defects was 9%, suggesting that increasing coverage has little effect on defect reduction. This finding is particularly important for companies that employ coverage measures

to decide upon test sufficiency. One of the findings of Mockus, et al. [59] was that increasing coverage linearly consumes effort exponentially. This finding emphasizes even more that increasing coverage alone is an inadequate technique for defect prevention. The key message from this finding is that practitioners should not focus on sheer fulfillment of the coverage criterion, but should apply more sophisticated techniques for unit testing to detect a maximal amount of defects with minimal testing effort. Exploratory testing [60] and causal analysis [61] are examples of such techniques.

The results of finding twelve deepen the scrutiny of coverage-defect relationship in the context of size, complexity, and evolution of code. The results clearly show that all of these properties have much larger effect on defects than coverage. In particular, the number of revisions, as shown earlier, has an effect size of 62%. One interesting aspect of complexity is that the *maximum level of nesting in a file* has a tangible effect size on both defects (16%) and coverage (18%). As this measure is defined not for a file, but for a block of code, it would be interesting to do further research and determine the extent to which nesting actually affects both defects and coverage. The results suggest that managing complexity can have much larger effect on defects than on managing unit test coverage. The results also tentatively suggest that managing nesting can have double the positive effects – decreasing defects and facilitating testing.

As Briand, et al. [27] rightfully observe, a measure is considered to be empirically valid if it can predict an external attribute (e.g., maintainability). Finding 13 suggests that empirical validation of complexity measures is not always possible because external attributes are not always measurable accurately. Statistical methods need to have historical data for both complexity measures and measures of external attributes in order to assess the predictive power of the complexity measures. Studies often use *defect count* as a dependent variable. Hence, complexity measures are usually evaluated by how well they predict defects. Yet counting defects for such entities as files and functions often cannot be done accurately; while a file can be considered defective, the actual root cause of the defect may be in another file yet since both files undergo defect correction activities, both are counted as defective. This means that defect count becomes inherently prone to inaccuracies. Consequently, studies report change measures to be better than static complexity measures in defect prediction, as in the case of ref. [56]. Other important dependent variables are *readability* and *understandability* of code and *ease of code integration*, although these variables have no direct measures so their empirical validation may be problematic.

The issue of accurate measurement even affects simple measures, such as cyclomatic complexity [10] and the coupling measures of Henry and Kafura [12]. It is not clear, for example, how function invocations should be counted when the same function is called several times with different parameters' list, i.e., should the function be considered the same function or a different function?

Finding 14 proposes a method based on action research principles for validating complexity measures even if the dependent variables are not measurable. This

method was developed as a consequence of Finding 13. The main concept is that a group of expert practitioners can provide valuable qualitative input on measured entities. Having obtained input on a sufficient number of data points (files, functions, requirements), general conclusions can be made on the extent to which measurement values indicate problems with external attributes. Furthermore, action research cycles can be used to improve measurement accuracy via "define-refine-redefine" action research cycles. Consequently, both the usefulness of measures and measurement accuracy can be evaluated using this method, which should be used with statistical methods for triangulating results and drawing more accurate conclusions.

5.2 Proactive Complexity Assessment in Continuous Software Development

Our measurements showed that the overall number of source functions in large products is tens of thousands. Complexity monitoring requires the continuous identification and manual checking of functions subject to substantial complexity increases over short time intervals (typically days or weeks). Finding 2 shows that only a few functions and files drastically increase in complexity over short periods of time. Typically, the number of such files and functions ranges from 0 to 10. The predominant parts of the functions and files have stable complexity. These results are particularly helpful for an organization with a dedicated person for code quality management because this person can feasibly monitor the complexity increase and conduct corrective actions.

The cost of fixing such defects discovered in late development phases or by customers can be multifold greater than if they were found in the development phase [62]. Similarly, the effort to improve the maintainability of code that was developed months ago can be multifold greater than if the defect was located and improved at the time of development. Code improvements that are conducted in response to external signals (e.g., tester/customer dissatisfaction) are reactive improvements. In contrast to reactive improvements, which typically succeed defect reporting, proactive improvements are conducted before a given piece of code is merged with the main product code. The results of Finding 4 represent a measurement system for the proactive identification of risky code areas. The main idea is that the measurement system allows the risk of defect-proneness and degradation of maintainability to be assessed before these negative aspects can actually manifest themselves later in the product life cycle. This allows developers to conduct corrective actions proactively, thereby mitigating the risk well before the code is merged within the main product code.

As with code, there are risks associated with poor requirements' specifications; a low internal quality of requirements increases the risk of late design modifications and ultimately causes project cost overruns [63]. The results of Finding 7 support conducting proactive reviews of textual requirements to prevent late design modifications. The results show that integrating formula (2) into local

requirement management tools enables proactive reviews. Every practitioner, and particularly requirements' analysts, can run the analysis based on formula (2) whenever they want to. Since this analysis can be run in the same environment as used for requirements' writing and usage, the administrative effort for the analysis is only a few seconds per review. As opposed to manual reviews, where the entire review process can take several weeks, automated reviews are performed within seconds. Moreover, when feedback is given "just in time" of writing the requirements, it is much easier for the writer to make improvements immediately.

Finding 10 emphasizes the necessity of code complexity management from the practitioners' perspective. It suggests that according to a practitioners' cumulative perception, code complexity increases the average maintenance time by a factor of 2.5 to 5. Thus, if it was possible to simplify a complex area of code, then maintenance time on that area could be reduced by a factor of 2.5 to 5, a substantial reduction. It is crucial, however, to understand the extent to which complexity can be reduced for a given piece of code based on the current complexity measures. Qualitative estimates described in papers one and two suggest that complexity can be reduced significantly. Another study not included in this thesis, but listed as the first in the "Additional Papers" section, provides a more meticulous understanding of the potential for complexity reduction. Research in this area, however, remains scarce and more work is needed in order to estimate the extent of complexity reduction more accurately.

5.3 Software Complexity Landscape

Finding 8 indicates that two out of the proposed eleven code characteristics, namely *nesting depth* and *lack of structure*, have a major influence on the increase of code complexity. Curiously, the most popular code complexity measures currently used do not measure nesting. Little attention has been given to the nesting aspect in literature. A nesting-based measure designed by Harrison and Magel [64] in 1981 is virtually unused in literature or in practice. Moreover, reflections on nesting are rare, the most well-known being critiques of cyclomatic complexity measure, which takes no account of the nesting [65], [66]. As an aspect of complexity, nesting has both representational and structural natures. Some degree of nesting is always required to organize interrelated decision statements. Yet deep nesting, a major source of complexity according to our results, can always be avoided by smart coding techniques, such as decomposing the block into separate functions, combining the conditional tests, and using early returns.

Compared to nesting, the second characteristic (lack of structure) is purely a representational characteristic. In Paper 4, lack of structure is defined as incorrect indentations, improper naming and not using the same style of coding for similar patterns of code. A measure that captures aspects of lack of structure was proposed by Buse and Weimer [67]; they evaluated the same measure and found that it had a good accuracy of agreement with manual assessors of

code readability. Lack of structure can always be avoided because it is mainly associated with how developers choose to write code. Nevertheless, the measure of Buse and Weimer has not gained any popularity. Finding 8 emphasizes strongly that, indeed, measures and supporting tools are needed to measure lack of structure and nesting of code.

Finding 9 shows that well-known code complexity measures [22] (pp. 335-429) are rarely used in practice for at least three reasons: 1) the well-known complexity measures do not capture the most influential complexity triggers; 2) single complexity measures alone are not effective in estimating maintainability and error-proneness – a combination of measures is needed; and 3) practitioners need research support to integrate complexity measures into companies in such a way that their use will take little administrative effort yet ensure maximal effect for decision support [68].

6 LIMITATIONS

This research has two important limitations, both of which are related to the ability to generalize these results, as discussed by Checkland and Holwell [38].

Firstly, complexity for the lowest abstraction levels of software was investigated in this thesis. As regards code and tests, the complexity of isolated blocks, functions, and files was investigated, whilst in the case of requirements, complexity of isolated requirements was investigated. All results were evaluated for the entities of lowest abstraction level. Yet complexity for higher abstraction levels, such as components and subsystems, and which can be termed "architectural complexity" also exist. Architectural complexity refers to the complexity that emerges from the architectural components and interactions. Since the methods and findings created in this thesis were not evaluated for architectural complexity, we do not know whether they can aid architectural complexity management.

Secondly, all software products used in this research were large, mature, and had a long history of development. They primarily belong to the sector of embedded systems, and were developed by C, C++, and Java programming languages. They also all belong to the Nordic software industry and thus contain elements of the Nordic software development culture. As these research results have all been obtained within this context, it is not clear whether other types of products would significantly impact these results.

7 FURTHER WORK

The results of this thesis open new research directions in the three areas that we investigated: complexity landscape, complexity assessment, and proactive complexity assessment in continuous software development.

Firstly, it is worthwhile to identify the complete list of software characteristics and evaluate their influence on complexity increase. Here, empirical studies

could elucidate the influence of such characteristics on complexity increase. There is also the option of using interdisciplinary research, where concepts of cognitive psychology could help to determine exactly how the human mind handles complexity. Previously, there was too much emphasis on investigating complexity as a pure system property. But complexity is profoundly connected with human perception. The first research paper in the "Other Publications" section is related to this subject.

Secondly, the aforementioned investigation might enable the design of more sophisticated complexity measures that would accurately embrace pragmatically all aspects of complexity. These measures would involve not only the structural aspects of complexity, but also representational and evolutional aspects. Furthermore, the extent to which complexity is reducible should be investigated. The straightforward way to do this would be to investigate which software characteristics are essential and which are accidental. Identifying accidental characteristics could clearly indicate a potential for complexity reduction. To date, the task of complexity reduction is poorly supported by scientific facts; rather, it is left to the ingenuity of software developers.

Thirdly, it is important to investigate how a good complexity measure can be standardized and used in practice. The challenges of integrating a good complexity measure into the developmental environment should be understood thoroughly. Moreover, practitioners need guidelines for turning a complexity measure into a complexity indicator so that they can use the complexity measure to simplify the software. Ultimately, the goal of a complexity measure is not the defect or maintainability prediction, but continuous software improvement.

What is more, the area of architectural complexity can be investigated. Of particular interest are which characteristics make architecture complex and how these can be measured. The ultimate goal of such research would be to create methods that allow architectural complexity to be assessed and reduced. Controlling complexity at all levels (requirements, code, and architecture) would allow practitioners to expand their software virtually beyond limits whilst keeping every composite part of software simple and maintainable.

PAPER 1

Monitoring Evolution of Code Complexity and Magnitude of Changes

ABSTRACT

Complexity management has become an indispensable activity in continuous software development. While the overall perceived complexity of a product increases rather insignificantly, the small units, such as functions and files, can have significant complexity increase with every increment of product features. This kind of evolution triggers risks of escalating defect-proneness and deteriorating maintainability. The goal of this research was to develop a measurement system that enables effective monitoring of complexity evolution. An action research has been conducted in two large software development organizations. Three complexity and two change measures of code were measured for two large industrial products. The complexity increase was measured for five consecutive releases of the products. Different patterns of growth have been identified and evaluated with software engineers in industry. The results show that monitoring cyclomatic complexity evolution of functions and number of revisions of files focuses the attention of engineers to risky files and functions for manual assessment and improvement. A measurement system was developed in one of the organizations to support the monitoring process.

1 INTRODUCTION

Actively managing software complexity has become an important activity of continuous software development. It is generally accepted that software products developed in a continuous manner are getting more and more complex over time. Evidence shows that the rising complexity drives to deteriorating maintainability of software [3, 69, 70]. The continuous increase of complexity can lead to virtually unmaintainable source code, if complexity is left unmanaged.

A number of measures were suggested previously to measure various aspects of software complexity and evolution over development time [71]. Those measures were accompanied with a number of studies indicating how adequately the proposed measures relate to software quality [66, 72]. Complexity and change measures have been used extensively in recent years for assessing the maintainability and defect-proneness of code [73]. However, despite the considerable amount of research conducted for investigating the influence of complexity on software quality, little results can be found on how to effectively monitor and prevent complexity growth. Therefore a question remains:

> *How can we monitor code complexity and changes effectively when delivering feature increments to the main code branch?*

The aim of this research was to develop a method and tool support for actively monitoring complexity evolution and drawing the attention of practitioners to the risky trends of growing complexity. In this paper we focus on the level of self-organized software development teams who often deliver code to the main branch for further testing, integration with hardware, and ultimate deployment to end customers.

We address this question by conducting an action research project in two companies, which develop software according to Agile and Lean principles. The studied companies are Ericsson which develops telecom products and Volvo Group Truck Technology (GTT) which develops electronic control units (ECU) for trucks.

Our results show that using two complementary measures, McCabe's cyclomatic complexity of functions and number of revisions of files supports teams in decision making, when delivering code to the main branch. The evaluation shows that monitoring trends in these measures helps identifying a handful of risky functions and files. These functions and files are manually assessed by the self-organized agile teams, who make decisions whether to refactor or to integrate the code to the main code branch.

2 RELATED WORK

Continuous software evolution: A set of measures useful in the context of continuous deployment can be found in the work of Fritz [74]. The metrics pre-

sented by Fritz measure aspects of continuous integration such as the pace of delivery of features to the customers. These measures complement the two indicators presented in this paper with business perspective which is important for product management.

The delivery strategy, which is an extension of the concept of continuous deployment, has been found as one of the three key aspects important for Agile software development organizations in a survey of 109 companies by Chow and Cao [75]. The indicator presented in this paper is a means of supporting organizations in their transition towards achieving efficient delivery processes.

Ericsson's realization of the Lean principles combined with Agile development was not the only one recognized in literature. Perera and Fernando [76] presented another approach. In their work they show the difference between the traditional and Lean-Agile way of working. Based on our observations, the measures and their trends at Ericsson were similar to those observed by Perera and Fernando.

Measurement systems: The concept of an early warning measurement system is not new in engineering. Measurement instruments are one of the cornerstones of engineering. In this paper we only consider automated measurement systems – i.e. software products used as measurement systems. The reasons for this are: the flexibility of measurement systems, the fact that we work in the software field, and similarity of the problems – e.g. the concept of measurement errors, automation, etc. An example of a similar measurement system is presented by Wisell [77], where the concept of using multiple measurement instruments to define a measurement system is also used. Although differing in domains of applications these measurement systems show that concepts which we adopt from the international standards (like [78]) are successfully used in other engineering disciplines. We use the existing methods from the ISO standard to develop the measurement systems for monitoring complexity evolution.

Lawler and Kitchenham [79] present a generic way of modeling measures and building more advanced measures from less complex ones. Their work is linked to the TychoMetric tool. The tool is a powerful measurement system, which has many advanced features not present in our framework (e.g. advanced ways of combining measures). A similar approach to the TychoMetric's way of using measures was presented by Garcia, et al. [80]. Both the TychoMetric tool and Garcia's tool provide advanced data presentation or advanced statistical analysis over time. Our research is a complement to [79] and [80]. We contribute by showing how the minimal set of measures can be selected and how the measurement systems can be applied regularly in large software organizations.

Mayer [81, pp. 99-122] claims that the need for customized measurement systems for teams is one of the most important aspects in the adoption of measures at the lowest levels in the organization. Meyer's claims were also supported by the requirements that the customization of measurement systems and development of new ones should be simple and efficient in order to avoid unnecessary costs in development projects. In our research we simplify the ways of

developing key performance indicators exemplified by a 12-step model of Parmenter [82] in the domain of software development projects.

3 DESIGN OF THE STUDY

This study was conducted using action research approach [48, 83, 84]. The researchers were part of the company's operations and worked directly with product development units. The development of the method and its initial evaluation was carried out at Ericsson, whereas the replication of the study was carried out at Volvo GTT.

3.1 Studied Organizations

Ericsson: The collaborating organization of Ericsson developed large products for mobile packet core network. The number of the developers was up to 150. Projects were executed according to the principles of Agile software development and Lean production system, referred to as Streamline development within Ericsson [85]. In this environment, different development teams were responsible for larger parts of the development process compared to traditional processes: design teams, network verification and integration, and testing.

Volvo GTT: The collaborating organization at Volvo GTT developed ECU software for trucks. The collaborating unit developed software for two ECUs and consisted of over 40 engineers, business analysts, and testers at different levels. The development process was in the transaction from traditional to Agile.

3.2 Units of Analysis

During the study we analyzed two products – software for a telecom product at Ericsson and software for two ECUs at Volvo GTT.

Ericsson: The product was a large telecommunication product comprised by over two million lines of code with several tens of thousands C functions. The product had a few releases per year with a number of service releases in-between them. The product was in development for a number of years.

Volvo GTT: The product was an embedded software system serving as one of the main computer nodes for a product line of trucks. It consisted of a few hundred thousand lines of code and about ten thousand C functions. The analyses that were conducted at Ericsson were replicated at Volvo GTT under the same conditions and using the same tools. The results were communicated with engineers of the software product after the data was analyzed.

At Ericsson the developed measurement system ran regularly whereas at Volvo the analysis was done semi-automatically, running the measurement system whenever feedback was needed for the practitioners.

3.3 Reference Group

During this study we had an opportunity to work with a reference group at Ericsson and an engineer at Volvo GTT. The aim of the reference group was to support the research team with expertise in the product domain and to validate the intermediate findings as prescribed by the principles of action research. The group interacted with researchers in bi-weekly meetings for over 8 months. At Ericsson the reference group consisted of a product manager, a measurement program leader, two engineers, one operational architect and one research engineer. At Volvo GTT we worked with one engineer.

3.4 Measures in the Study

Table 1 presents the complexity measures, change measures and changes (deltas) of complexity measures over time. The definitions of measures and their deltas are provided in the table.

Table 1 Measures and definitions

Complexity Measures	**Abbr.**	**Definition**
McCabe's cyclomatic complexity of a function	M	The number of linearly independent paths in the control flow graph of a function, measured by calculating the number of 'if', 'while', 'for', 'switch', 'break', '&&', '||' tokens
Structural Fan-out	Fan-out	The number of invocations of functions found in a specified function
Maximum Block Depth	MBD	The maximum level of nesting found in a function
M of a file	M_f	The sum of all functions' M in a file
Change Measures	**Abbrev.**	**Definition**
Number of revisions of a file	NR	The number of check-ins of files in a specified code integration branch and its all sub-branches in a specified time interval
Number of engineers of a file	ND	The number of developers that do check-in of a file on a specified code integration branch and all of its sub-branches during a specified time interval
Deltas of Complexity Measures	**Abbrev.**	**Definition**
Complexity deltas of a function	ΔM ΔFan-out ΔMBD	The increase or decrease of M, Fan-out and MBD measures of a function during a specified time interval. We register the file name, class name and function name in order to identify the same function and calculate its complexity change over releases.

3.5 Research Method

We conducted the study according to the following pre-defined process. The first seven steps are conducted on both Ericsson and Volvo products. The rest of the steps are carried on for the Ericsson product.

1. Obtain access to the source code of the products and their different releases
2. Measure complexity of all functions and changes of all files of the code
3. Measure complexity deltas of all functions and changes of all files for the five releases of the products
4. Sort the functions by complexity delta and sort the files by change magnitude through the five releases
5. Identify possible patterns of complexity deltas and changes
6. Identify drivers and possible explanations for the highest complexity deltas and the highest magnitude of changes
7. Correlate measures to explore their dependencies and select measures for monitoring complexity and changes
8. Develop a measurement system (according to ISO 15939) for monitoring complexity and changes
9. Monitor and evaluate the measurement system for five weeks
10. The overall complexity change of function is calculated by: Overall delta = $(\Delta M_{rel12})+(\Delta M_{rel23})+(\Delta M_{rel34})+(\Delta M_{rel45})$ where (ΔM_{relij}) is the value of McCabe complexity change of a function between i and j releases.

Overall complexity change of Fan-out and MBD is calculated the same way.

4 ANALYSIS AND RESULTS

In this section we explore the main scenarios of complexity evolution. We carry out correlation analysis of collected measures in order to understand their dependencies and delect measures for monitoring.

4.1 Evolution of the Studied Measures over Time

Exploring different types of changes of complexity, we categorized changes into 5 groups. The five groups are described in the following five points:

1. Functions that are newly created and become complex in current release and functions that were complex but disappeared in current release
2. Functions that are re-implemented in the current release
3. Functions that have significant complexity delta between two releases due to development or maintenance

4. Test functions, which are regularly generated, destroyed and regenerated for unit testing
5. Functions that have minor complexity changes between two releases

Group 1 and group 5 functions were observed to be the most common. They appeared regularly in every release. Engineers of the reference group characterized their existence as expected result of software evolution. Group 2 functions were re-implementation of already existing function. The existed functions were re-implemented with different name and the old one was destroyed. After re-implementation the new functions could be named as the old one. Re-implementation usually took place when major software changes were happening: In this case re-implementation of a function sometimes could be more efficient than its modification.

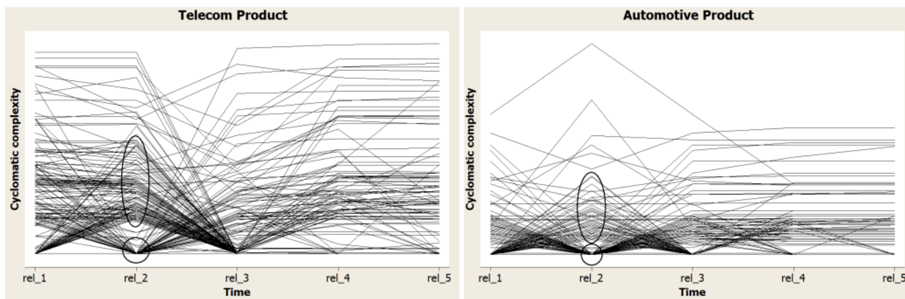


Figure 1 Evolution of M of functions

Figure 1 shows the cyclomatic complexity evolution of top 200 functions through the five releases of the products. Each line in the figure represents a C function. In Figure 1 re-implemented functions are outlined by elliptic and old ones by round lines. In reality the number of re-implemented functions is small (about 1 %), however, considering the big complexity deltas of them, many of them ended-up in the top 200 functions in the figure, giving an impression that they are relatively many. Figure 2 similarly presents the evolution of Fan-out in the products. Group 3 functions are outlined by elliptic lines in Figure 2.

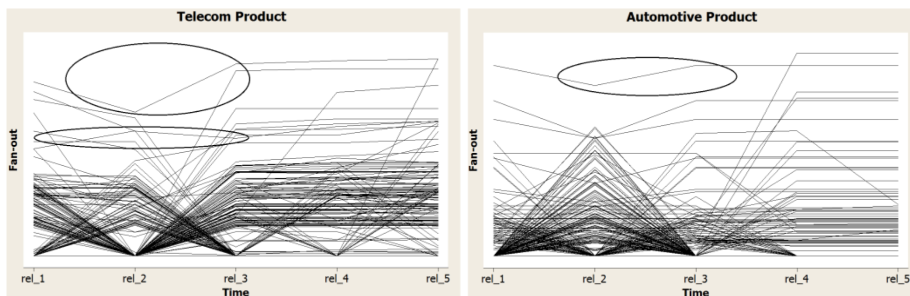


Figure 2 Evolution of Fan-out of functions

Group 3 functions were usually designed for parsing a huge amount of data and translating them into another format. As the amount and type of data is changed the complexity of the function also changes. Finally the Group 4 functions were unit test implementations. These functions were destroyed and regenerated frequently in order to update running unit tests. Figure 3 presents the MBD evolution of products. As nesting depth of blocks can be obtain much more limited values, many lines in Figure 3 overlap each other thus creating an impression that there are few functions. We observed that the functions of group 1, ones were created, stayed complex over time. These functions are outlined with a rectangular line in Figure 3.

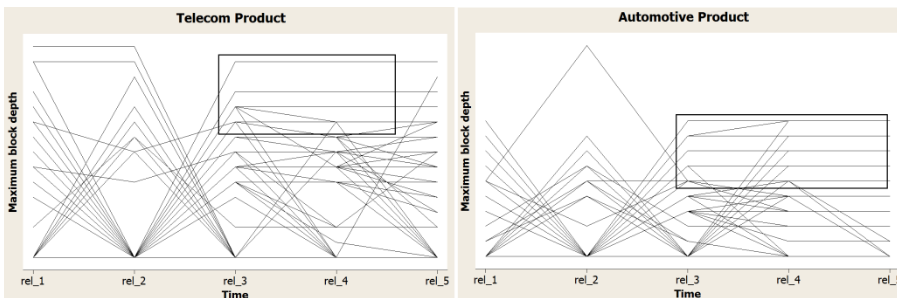


Figure 3 Evolution of MBD of functions

The proportions of all functions percentagewise represented in Table 2. The table shows how all functions, that had complexity change, are distributed in groups. We would like to mention that the number of all functions in telecom product is about 65000 and in automotive product about 10000, however only top 200 functions out of those are presented in the figures. This might result in disproportional visual understanding of the relation between different groups of functions in the table and in the figures, as the figures contain only top 200 functions.

Table 2 The distribution of functions with complexity delta in groups

Group	Group 1	Group 2	Group 3	Group 4	Group 5
Percentage	27%	1%	1%	1%	70%

We observed the deltas of complexity for both long time intervals (between releases) and for short time intervals (in weeks). Figure 4 shows how the complexity of functions changes over weeks. The initial complexity of functions is provided under column M in the figure.

File name	Function name	M	Total: ΔM	w1306	w1307	w1308	w1309	w1310	w1311	w1312
file 1	function 1	14	0	0	0	0	0	0	0	0
file 2	function 2	15	15	0	0	0	0	0	15	0
file 2	function 3	1	0	0	0	0	0	0	0	0
file 3	function 4	10	5	4	-9	11	-11	10	0	0
file 4	function 5	11	3	0	0	0	0	11	0	0
file 5	function 6	58	13	17	0	11	-11	0	0	-4
file 5	function 7	22	22	0	0	0	0	0	0	22
file 6	function 8	20	20	0	0	0	18	2	0	0
file 6	function 9	17	17	0	0	0	17	0	0	0
file 7	function 10	11	11	0	0	0	11	0	0	0
file 8	function 11	13	13	0	0	0	0	13	0	0
file 9	function 12	28	28	0	28	0	0	0	0	0
file 10	function 13	12	12	0	0	0	12	0	0	0

Figure 4. Visualizing complexity evolution of functions over weeks

The week numbers are presented on the top of the columns, and every column shows the complexity delta of the functions in that particular week. Under ΔM column we can see the overall delta complexity per function that is the sum of weekly deltas per function.

The fact that the complexity of the functions fluctuates irregularly was interesting for the practitioners, as the fluctuations indicate active modifications of functions, which might be due to new feature development or represent defect removals with multiple test-modify-test cycles. Functions 4 and 6 are such instances illustrated in Figure 4. Monitoring the complexity evolution through short time intervals we observed that very few functions are having significant complexity increase. For example, in a week period of time the number of functions that have complexity increase $\Delta M > 10$ are not more than ten.

4.2 Correlation Analyses

The correlation analyses of measures were carried out to eliminate dependent measures and select a minimal set of measures for monitoring. The correlation analyses results are presented in Table 3. The plot of the relationship of the complexity measures is presented in Figure 5. As the table illustrates there is a strong correlation (0,76) between M and Fan-out measures for the telecom product, while the correlation between the same measures is rather weak for the automotive product (0,26). This means that only M measure is enough to be monitored in the telecom product, because it also encompasses most of the information that could be obtained from the Fan-out measure.

Table 3 Correlation of complexity measures

Telecom / **Automotive**	MBD	M
M	0,41 / **0,69**	
Fan-out	0,34 / **0,20**	0,76 / **0,26**

Generally, this also means that the correlation between these two measures can vary greatly from product to product, and for every product a correlational analysis should be carried out to find out whether these measures are correlated. The correlation between M and MBD for the automotive product is also strong (0,69), while the correlation between M and MBD measures for the telecom product is moderate (0,41). Generally the discussions with the reference group led us to understanding that monitoring cyclomatic complexity among all complexity measures is good enough as there was a moderate or strong correlation between the three complexity measures. M was chosen because of two reasons:

1. MBD is rather a characteristic of a block of code than a whole function. It is a good complementary measure but it cannot characterize the complexity of a whole function.
2. Fan-out seemed to be a weaker indicator of complexity than M, because it rather shows the vulnerability of a function due to its dependence on other functions.

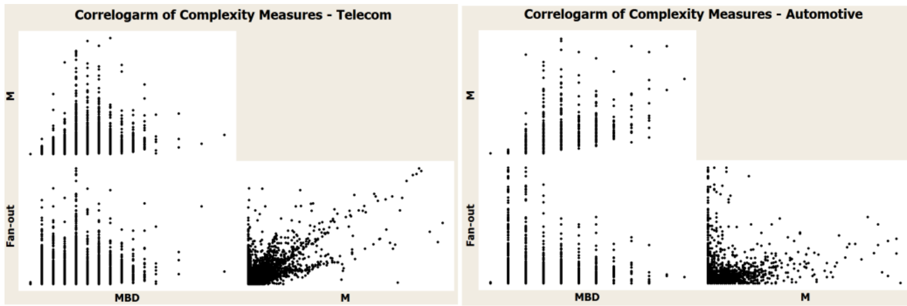


Figure 5 Correlograms of complexity measures

NR and ND are measures that indicate the magnitude of changes. Previously a few studies have shown that change measures are good indicators of problematic areas of code, as observed by Shihab, et al. [86]. The measurement entity of NR and ND is a file. Therefore in order to understand how change measures correlate with complexity measures we decided to define a cyclomatic complexity measure for files (Table 1). Table 4 presents the correlation analysis results for ND, NR and M_f measures.

An important observation is the strong correlation between the number of designers (ND) and the number of revisions (NR) for the telecom product. At the beginning of this study the practitioners of the reference group believed that a developer of a file might check-in and check-out the file several times which probably is not a problem.

Table 4 Correlation of change and complexity measures

Ericsson / **Volvo**	M_f	ND
ND	0.40 / **0.37**	
NR	0.46 / **0.72**	0.92 / **0.41**

The real problem, they thought, could be when many designers modify a file simultaneously. Nonetheless, a strong correlation between the two measures showed that they are strongly dependent, and many revisions is mainly caused by many engineers modifying a file in a specified time interval (Figure 6).

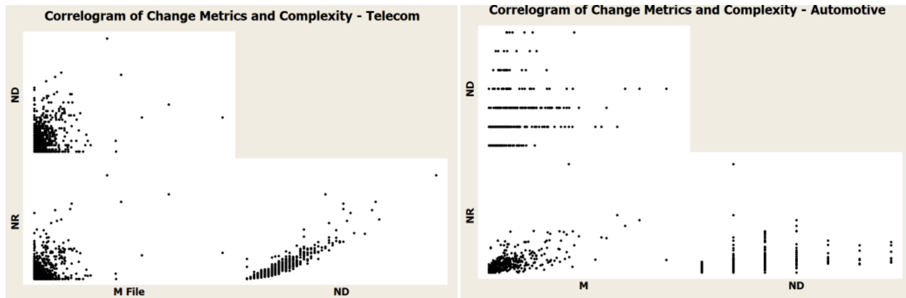


Figure 6 Correlograms of change and complexity measures

In case of the automotive product the correlation between ND and NR was moderate which can be due to small number of engineers who have rather firmly assigned development areas and usually change the same code. The explanation of the strong correlation between M_f and NR is more complicated. As we reflected on M_f measure, (which is also used in other studies), in many cases it indicates more size and less complexity of a file. A file may be composed of many small functions with small cyclomatic complexity numbers. The sum of all complexities, however, can build up into a large number file with a big M_f number. Then the correlation between M_f and NR can be explained by pure probabilistic reasons – larger files are more likely to be changed. Therefore, since M_f is not a well-motivated measure, we decided to monitor the number of revisions on a file level.

The results showed that for telecom product only NR and M measures need to be monitored, because they contain most of the information that all the measures would provide collectively. Considering these results we designed a measurement system at Ericsson for monitoring code complexity and magnitude of changes over time. The description of design and application of measurement system is discussed in the next section.

4.3 Design of the Measurement System

We designed two indicators based on M and NR measures. These indicators capture the increase of complexity of the functions and highlight the files with highest magnitude of change over time. These indicators were designed according to ISO/IEC 15959 standard. The design of complexity indicator is presented in Table 5.

Table 5 Measurement system design based on ISO/IEC 15939 standard

Information Need	Monitor cyclomatic complexity evolution over development time
Measurable Concept	Complexity delta of delivered source code
Entity	Source code function
Attribute	Complexity of C functions
Base Measures	McCabe's Cyclomatic complexity number of C functions – M
Measurement Method	Count cyclomatic number per C function according to the algorithm in CCCC tool
Scale	Positive integers
Unit of measurement	Execution paths over the C/C++ function
Derived Measure	The growth of cyclomatic complexity number of a C function in one week development time period
Measurement Function	Subtract old cyclomatic number of a function from new one: $\Delta M = M(week_i) - M(week_{i-1})$
Indicator	Complexity growth: *The number of functions that exceeded McCabe complexity of 20 during the last week*
Model	Calculate the number of functions that exceeded cyclomatic number 20 during last week development period
Decision Criteria	If the number of functions that have exceeded cyclomatic number 20 is different than 0 then it indicates that there are functions that have exceeded established complexity threshold. This suggests the need of reviewing those functions, finding out the reasons of complexity increase and refactoring if necessary

The other indicator based on NR is defined in the same way: the files that had NR > 20 during one week development time should be identified and reviewed. The measurement system was provided as a gadget with the necessary information updated on a weekly basis (Figure 7). The measurement system relies on a previous study carried out at Ericsson [87, 88].

For instance the total number of files with more than 20 revisions since last week is 5 (Figure 7). The gadget provides the link to the source file where the engineers can find the list of files or functions and the color-coded tables with details (see Figure 4).

As in agile development the development teams merge builds to the main code branch in every week it was important for the teams to be notified about functions with drastically increased complexity (over 20).

No of Complex functions

2013-08-26
No of func. ΔM > 20
2
No of func. NR > 20
5
Source data

Figure 7 Information product for monitoring ΔM and NR measures over time

5 THREATS TO VALIDITY

The main external validity threat is the fact that our results come for an action research. The research on two cases indicated that the results can vary greatly from company to company, therefore to develop a measurement system, one needs to conduct the analysis of this paper but make decisions based on the specific results.

The main internal validity threat is related to the construct of the study and the products. In order to minimize the risk of making mistakes in data collection we communicated the results with reference groups at both companies to validate them.

The threshold 20 for cyclomatic number does not have any firm empirical or theoretical support. It is rather an agreement of developers of large software systems. We suggest that this threshold can vary from product to product. The number 20 is a preliminary established number taking into account the number of functions that can be handled on a weekly basis by developers.

The main construct validity threats are related to how we identify the names of functions for comparing their complexity numbers over time. There are several issues emerging in this operation. Namely, what happens if a function has changed its list of arguments or what happens if a function is moved to another file? Should this be regarded as the same function before and after changing the list of arguments or the position? We disregarded the change of argument list however this can be argued.

Finally the main threat to conclusion validity is the fact that we do not use inferential statistics to monitor relation between the code characteristics and project properties, e.g. number of defects. This was attempted during the study but the data in defect reports could not be mapped to individual files. This might be a thread for jeopardizing the reliability of such an analysis. Therefore we chose to rely on the most skilled engineers' feedback on what a good measure is.

6 Conclusions

In this paper we explored how complexity evolves, by studying two software products – one telecom product at Ericsson and one automotive product at Volvo GTT. We identified that in short periods of time a few out of tens of thousands functions have significant complexity increase.

By analyzing correlations between three complexity and two change measures we concluded that it is enough to use two measures, McCabe complexity and number of revisions for one of the products. This measurement can draw the attention of practitioners to risky code areas for review and improvement.

The automated support for the teams was provided in form of a gadget with the indicators and links to statistics and trends with detailed data on complexity evolution. The measurement system was evaluated by using it on an ongoing project and communicating the results with practitioners.

PAPER 2

Identifying Risky Areas of Source Code in Agile Software Development

ABSTRACT

Modern software development relies on incremental feature delivery to facilitate quick response to customers' requests. In this dynamic environment the continuous modifications of software code can trigger risks for software developers: When developing a new feature increment, the added or modified code may contain fault-prone or difficult-to-maintain elements. The outcome of these risks can be defective software or decreased development velocity. This study presents a method to identify the risky areas and assess the risk when developing software in agile environment. We have conducted an action research project in two large companies, Ericsson and Volvo Group Truck Technology. During the study we have measured a set of code properties and investigated their influence on risk. The results show that the superposition of two measures can effectively enable identification and assessment of the risk. We also illustrate how this kind of assessment can be successfully used by software developers to manage risks on a weekly basis as well as release-wise. A measurement system for systematic risk assessment has been introduced to two companies.

1 INTRODUCTION

Increasing complexity of modern software products has become a well-known problem. Escalating fault-proneness and declining maintainability of software are the main risks behind this kind of increase. Due to increasing size of software products and the need for increased development velocity the traditional risk assessment methods [89-94] are not applicable in identifying and assessing these kind of risks. For example it is impossible for an expert to identify the most difficult-to-maintain files out of several thousands in a product, whereas this kind of assessment is needed on a regular basis for supporting practitioners in systematic mitigation of risks.

Several studies have shown that the code is continuously becoming more complex if left unmanaged [3, 69, 70], and with growing complexity momentous technical risks emerge. Fenton and Neil [68] claim that technical risk assessment is essential for supporting software engineers in decision making, yet most of the studies in the field are concentrated on a narrower field – defect predictions [95-102]. Despite the importance of other aspects of risks than fault-proneness, very few researchers have proposed methods for full risk identification and assessment that is adopted by industry. Therefore an open question remains:

> *How can we effectively identify risky source code and assess the risk when delivering new feature increments in agile development?*

In this context we define the *risk* as likelihood that a source code file becomes fault prone, difficult-to-manage or difficult-to-maintain. Manageability of the code is concerned with such activities as assigning certain areas of code to certain developers, merging the code to the main code base, and controlling different variants of code for different customer groups.

The aim of this study was to develop a method and supporting tool for enabling systematic identification and assessment of risks, when delivering new code in agile production. To address this question we designed and conducted an action research project together with Ericsson and Volvo Group Trucks Technology (Volvo GTT).

We created a method and supporting tool for identification of risky files. The method is based on measurement of two properties of code: the revisions and complexity of a file. We evaluated the method in an industrial context by applying it on ongoing software development projects. The evaluation of the method showed that all severe risks were identified, and the method helped the engineers and architects to focus on about 0.1% of the code base, which are the risky files. The assessment method was evaluated to be effective for its application in industry.

2 AGILE SOFTWARE DEVELOPMENT

Agile software development in large companies is characterized by a combination of challenges of large products, such as, long development cycles, long-term release planning, distributed decision making by software development teams, communication between teams, etc. Figure 1 presents an overview on how the functional requirements (FR) and non-functional requirements (NFR) are packaged into work packages (WP) and developed as features by the teams. Each team delivers their code into the main branch. Each team has the possibility to deliver code for any component of the product.

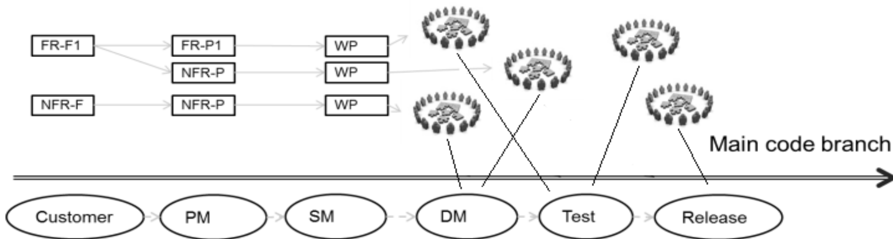


Figure 1 Feature development by agile methodology.

First, the requirements come from the customers, and are prioritized and packaged into features by product management (PM). Next, PM hands over the requirements to the system management (SM) for systemization. Then, design management (DM) and test teams implement and verify them before delivering to the main branch. Last, the code in the main branch is additionally tested by dedicated test units before the release [103].

In this context software development is a continuous activity, with small increments on a daily or weekly basis to a large code base, which exists over long periods of time. In order to manage the risk of degrading maintenance in continuous development the risk management also needs to be a continuous activity.

3 STUDY DESIGN

We applied an action research method in our study by maintaining close collaboration with industry practitioners and regularly working at the companies' premises.

3.1 Industrial Context

At Ericsson, the organization where the research was conducted develops large products for mobile telephony network. Several hundred developers comprise the development organization. Projects are conducted according to the princi-

ples of agile software development and lean production procedures, called streamline development in Ericsson [85]. In this environment cross-functional development teams are responsible for accomplishing a set of development activities: analysis, design, implementation, and testing assigned features of the product.

To support managers, designers, and quality managers in decision making, a measurement organization was established at Ericsson (7 years before writing of this paper) for calculating and presenting variety of indicators and early warning systems.

The developed unit was a large telecom product which constituted a few millions of lines of code with several thousands of C/C++ files. The product was released a few times per year with support of service releases. Rational ClearCase served as version control system by which all the source code of the product was handled. The product had been in development for more than 15 years.

At Volvo Group Trucks Technology (GTT), the organization in which we have worked developed software of Electronic Control Units' (ECU) for Volvo, Renault, UD Trucks and Mack. Our collaborating unit developed software for an ECU which consists of a few hundred thousand lines of code and more than one thousand files entirely developed by C language. The product was released in every 6-8 weeks. About 50 designers, business analysts and testers comprised the organization. The development process was progressing toward agile development.

The organization systematically uses various measures to control the progress of development and monitor the quality of the products. Our intention was to develop a method and tool for risk assessment of in-house developed software as well as outsourced and imported software.

3.2 Reference Groups at the Companies

During this study we had the opportunity to work with a reference group initiated at Ericsson. The reference group was to support the research team with expertise in the product domain and to scrutinize and reflect on intermediate findings. The group meetings took place on bi-weekly basis for over 8 months. The reference group consisted of one line manager, one measurement program leader, two designers, one operational architect and one research engineer. At Volvo we worked with a designer and a line manager.

3.3 Flexible Research Design

Five main research cycles were carried out:

1. Identification of measures: Shortlisting a number of published measures which can theoretically be used in risk assessment

2. Measurement and analyses: Collecting data for these measures and analyzing inter-dependencies of the measures
3. Creation of risk identification and assessment method: Developing a method based on the selected measures
4. Evaluation of the method with engineers: Evaluating the method over a number of weeks through by-weekly meetings with reference group, and according to the empirical measurement validation principles [104].
5. Refinement and evaluation of the method in the projects: Disseminating the method to all engineers in the project and monitoring the use of the method.

The planned cycles above were concretized during the study. Thus we carried out the following steps to fulfill above defined five cycles:

1. Obtain access to the source code of the products and their different releases: Decided upon one product per company, releases of the product, whether service releases should be included or not
2. Set up necessary tools for extracting data: Develop scripts for data collection in Ruby, MS Excel VBA.
3. Calculate code measures per defined entities (files\functions)
4. Carry out calculation for four releases of the products
5. Identify drivers of high complexity/change through interviews with engineers and the reference group
6. Correlate measures to explore their relations and determine which measures should be selected
7. Develop a method by using the selected measures for identifying and assessing the risks, and establish decision criteria for determining the risk exposure per file
8. Identify the risky files and assess the risk using the method and decision criteria
9. Collect post-release error reports (ER) per file for four service releases
10. Evaluate the method by:
    a. Correlating the ERs and calculated risk exposure of files
    b. Assigning files to responsible designers for manual assessment (6 weeks period)
11. Develop a measurement system according to ISO 15939 to manage the risky files [105]

The above process was used during the development of the method at Ericsson and replicated at Volvo GTT.

3.4 Definition of Measures

In order to assess the risky code we used nine measures of code described in Table 7. These measures measure such properties of code as size, complexity, dependencies and change frequency. The choice of properties was motivated by:

1. How well these properties of code can be predictors of risk (identified from existing literature)
2. Which properties can relate to risk according to the perception and experience of the reference group

Table 1 presents the measures of code properties which we used in our study and their definitions. *It was not possible to measure ND and NR for functions so we measured them only for files*. Other properties that were defined for functions were possible to redefine for files also.

Table 1 Measures and definitions

We defined and measured M and NCLOC for files so we could correlate these

Name of measure	Abbrev.	Definition
Number of non-commented lines of code	NCLOC	The lines of non-blank, non-comment source code in a function/file (this property is measured for both units)
McCabe's cyclomatic complexity of a function	M	The number of linearly independent paths in the control flow graph of a function, measured by calculating the number of 'if', 'else', 'while', 'for', '||', '&&', 'switch', 'break', 'goto', 'return' and 'continue' tokens
McCabe's cyclomatic complexity of a file	File_M	The sum of all functions' M in a file
Structural fan out of a function	Sfout	The number of function calls found in a specified function
Max block depth	MBD	The depth of max nested block in a function
Number of revisions of a file	NR	The number of check-ins of a file in a specified ClearCase branch and all its sub-branches in a specified time interval
Number of designers of a file	ND	The number of developers that do check-in of a file on a specified ClearCase branch and all of its sub-branches during a specified time interval
Effective cyclomatic complexity of a file	M_{ef}	The complexity sum of all functions with M > 15 in a file
Effective cyclomatic complexity percentage of a file	$M_{ef\%}$	The ratio of M_{ef} and File_M This measure shows how much of the complexity of a file composed by complex function

measures with ND and NR, and understand their relation. Correlation analysis was carried out as a necessary step for determining which measure of code to choose for risk prediction. Collinear measures most likely indicate the same property of code. It is important to notice that correlation analyses were not sufficient for selecting measures so further analysis was also carried out to understand other aspects of measures' relations. Later in the study, during the evaluation with designers we found that it is important to distinguish between files with many small non-complex functions and files with a few large complex functions. Thus we defined the following measures, M_{ef} and $M_{ef\%}$ presented in Table 1. The aim of the $M_{ef\%}$ is to show what portion of the File_M number is distributed in complex functions of a file.

We calculate these two measures the following way:

$$\boldsymbol{M_{ef} = \sum_{i=1}^{n} M_i \ \ for\ all\ M_i > 15\ in\ a\ file} \qquad \text{(Eq. 1)}$$

$$\boldsymbol{M_{ef\%} = \left(\frac{M_{ef}}{File_M}\right) * 100} \qquad \text{(Eq. 2)}$$

The functions having M > 15 are considered complex. In his paper McCabe [106] defines a threshold for M as 10. However, considering the fact that there are other suggested limits like 15 and 20, we chose 15. An example of how to calculate $M_{ef\%}$ is shown in Table 2. As the table illustrates, for the specified file the functions 1, 3 and 5 are complex as they have M > 15 complexity.

Table 2 An example of calculating $M_{ef\%}$

File	Function	M	File M	M_{ef}	$M_{ef\%}$
<u>file.c</u>			<u>82</u>	<u>58</u>	<u>71%</u>
	function 1	21	= 8+0+5+17+3+20+8+21	= 17 + 20 + 21	= (58 / 82)*100%
	function 2	8			
	function 3	20			
	function 4	3			
	function 5	17			
	function 6	5			
	function 7	0			
	function 8	8			

Thus the M_{ef} of this file is the sum of complexities of functions 1, 3 and 5, which is 58. Dividing this number by overall complexity File_M we get $M_{ef\%}$ = 71%.

This kind of representation of complexity for a file is more informative and appreciated by software designers as it does not ignore the fact that functions are independent units. It is importance not only quantifying the complex portion of file but also not losing relatively small files that contain complex functions. For example, if file A has only 2 functions, both of them having M = 25, then the sum

of complexities of the file would be 50. Large files having many simple functions and significantly larger File_M should not in reality be considered more complex than file A. $M_{ef\%}$ holds an ability to show that file A is complex irrespective its size because it contains complex functions: $M_{ef\%}$ = 100%.

4 RESULTS

In this section we present the correlation analysis of measures, designers' comments on correlations of measures, the created method for risk assessment, and the established thresholds.

4.1 Correlation Analysis

Correlation analyses were used to select the necessary set of measures for risk assessment. Table 3 shows the correlations of 4 measures for both products.

Table 3 Correlation matrix of file measures

Ericsson / **Volvo**	NCLOC	File_M	ND
File_M	0.91 / **0.90**		
ND	0.47 / **0.38**	0.41 / 0.**40**	
NR	0.55 / **0.61**	0.48 / **0.68**	0.92 / **0.46**

As the table shows, the correlation between M and NCLOC is strong, 0.91 / 0.90 for telecom and automotive software conformably. However the McCabe's complexity originally is defined for functions thus high complexity number for files can be caused by summing the complexity numbers of many moderately complex, but unrelated functions in a file. The correlation between NR and NCLOC is moderate, 0.55 / 0.61 for telecom and automotive respectively. The existing moderate correlation is driven by the size of files: Bigger files are more likely to get changes during development. By the observation of designers and us there are two fundamentally different reasons behind complex files that are changed often and simple files that are changed often:

1. Simple files that are changed often are usually files that are in the core of development in a particular one or two week time period. They are not regarded risky as they are easy to understand and maintain and are not fault prone
2. Complex files changed often are predominantly files that contain complex functions and are executing complex tasks. These files are hard to understand and maintain and usually are changed periodically.

Initially the designers suggested that a good measure that would reveal the risky files is the number of designers (ND) making simultaneous changes on a file. The assumption was that the high number of revision can be achieved by

few developers also, when they work intensively on a file and do several check-ins to version control system. However, the strong correlation between NR and ND (0.92) for telecom software shows that high number of revisions is a result of many designers checking in and out a file. In Figure 2 and 3 the data points in the scatter plots are the files.

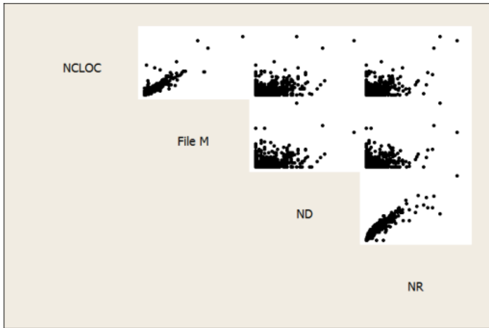


Figure 2 Correlogram of measures of files telecom software

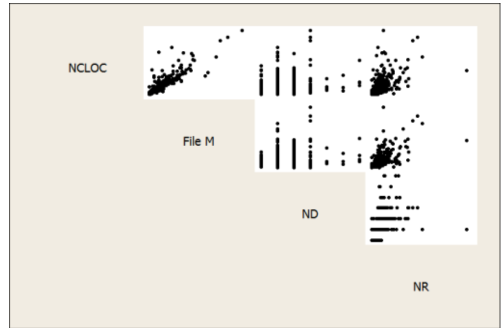


Figure 3 Correlogram of measures of files automotive software

Overall the correlations of measures are similar for both of the products, yet with an essential difference: the correlation between NR and ND is weak for the automotive software. The main reason is that the number of designers of ECU development team is much smaller than it is for the telecom software. Tens of designers distributed in cross-functional teams in telecom product virtually are available to be assigned various development tasks in different parts of the software, whilst every designer of ECU development team has rather assigned area of functionality to develop. In Figure 3 we can see that the data points for ND graphs are portioned like discrete lines indicating scarce number of designers.

Next we correlated the function measures for both of the products. Table 4 presents correlations between the function measures. It is important to notice that the correlation coefficient between M and NCLOC diminished significantly. Previously several studies has reported observed linear relationship between NCLOC and M [66, 107]. We confirm that there is significant correlation between these two measures, however we argue that NCLOC most likely cannot be a substitute for M measure and a further analysis is needed to understand their relationship in a deeper sense. The thorough examination of these two measures is out of the scope of this study but there are a few valuable observations worth to mention.

Table 4 Correlation matrix of function measures

Ericsson/**Volvo**	M	NCLOC	MBD
NCLOC	0.75 / **0.77**		
MBD	0.41 / **0.60**	0.24 / **0.44**	
Sfout	0.75 / **0.17**	0.87 / **0.77**	0.34 / **0.12**

Firstly, high cyclomatic complexity necessitates large functions. This implies certain positive correlation between these two measures. However, big functions are not necessarily complex. This simple observation is well expressed in correlograms of Figure 4 and Figure 5: Scatter plots of (M, NCLOC) show that the crowded data have a triangular shape.

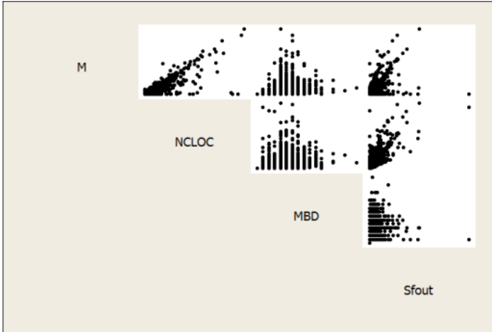


Figure 4 Correlogram of function measures for telecom software

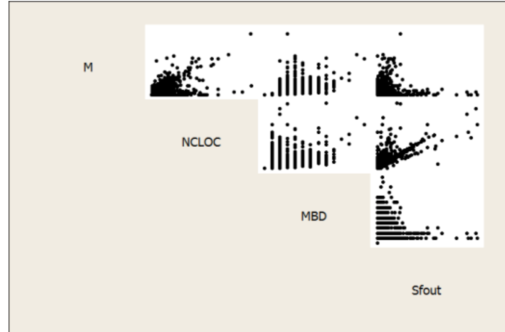


Figure 5 Correlogram of function measures for ECU software

This means that there are functions along the line of NCLOC axis showing 0 complexities but there is no function with high complexity and 0 NCLOC. Secondly, the correlation is calculated between NCLOC and M cyclomatic complexity numbers. But in reality the cyclomatic complexity number is intended to measure the complexity property. This means that the complexity that designers perceive and the complexity that cyclomatic number shows are different. And the relationship between the real complexity and the cyclomatic complexity number is not linear itself.

The correlation between Sfout and M is strong for telecom product, 0.75, whereas it is insignificant for ECU software, 0.17. The reference group designers at Ericsson believe that strong correlation between M and Sfout is product specific. They checked the code and identified that in telecom software functions usually execute “check message” - “call function” types of operations. We can observe also that (M, Sfout) scatterplot for telecom product is similar to (M, NCLOC) scatterplot: The data distribution has triangular shape. *This means that functions with high M number necessarily have high Sfout.* However the causality of these two measures is more complicated. For the automotive software M and Sfout have rather weak dependency. In Figure 5 the scatterplot of (M, Sfout) shows that several functions have high M number and low Sfout. Those are state machines. A few other functions have high Sfout numbers and low complexity. These functions are test code. The rest of the data is scattered over the graph and there is no specific pattern. According to the contact designer at Volvo, the functions that have high Sfout and low M numbers are not perceived to be complex. Certainly Sfout introduce complexity to the whole product, as it indicates more interconnections between functions but it is not a clear complexity meas-

ure for a single function. A function with high Sfout is rather vulnerable to the called functions.

The correlation between MBD and M is moderate for both products – (0.41 / 0.60). Nesting is an interesting property, but one problem is that it is not appropriately defined for a function or a file, but rather for a block of code. This makes it difficult to apply for decision making.

4.2 Selecting Measures

Based on correlation analysis and designers interpretation of results one change measure (revisions) and one complexity measure (cyclomatic complexity) were decided to be used for risk identification and assessment. The intuition behind this choice is as following:

1. Complexity makes the code hard-to-maintain, hard-to-understand and fault-prone.
2. The risk is triggered when there is a change in complex code.

As a change measure we selected NR. Firstly several studies has shown that high NR number is an indication of defect prone and difficult-to-maintain code [108, 109]. Secondly, NR and ND were correlated strongly for telecom product showing the same aspect of code, and finally, in case of automotive software there was no evidence that high ND indicates any tangible risk.

As a complexity measure we selected the cyclomatic complexity. Firstly this is a measure characterizing rather inner complexity of a function than the complex interactions that it has with the rest of the code: Complex interactions do not imply that the function itself is complex, but show how vulnerable the function is when making modifications in other parts of code. Secondly in the telecom product high cyclomatic complexity entails high Sfout number of functions (Figure 4, right uppermost plot).

MBD was perceived to be a good complexity measure, as it involves cyclomatic complexity and contains additional complexity also (nesting complexity). But it is hard to draw conclusions based on MBD. The reason is that MBD is defined for a block of code so it does not characterize the complexity of whole function. In order to use MBD with other measures it should be defined for the same code unit as other measures are defined.

4.3 Evaluation with Designers and Refinement of the Method

Our intention was to use M and NR for identifying risky files. However the two measures were defined for different entities of code: NR was defined for files and M for functions. *We determined the risk assessment unit to be a source code file*. NR was intended to be used in risk assessment but it was hard to define and

calculate number of revisions (NR) of a function so we had to select a file as risk assessment unit. We attempted to identify the risky files by selecting the files which have both high NR and File_M number. But evaluation with designers showed that File_M is not a good complexity measure. The reason is if many simple functions are placed in the same file then the sum of complexities might have a high value which is misleading. Instead if we calculate the average complexity per file as complexity sum divided by the number of functions, we get a number which might not show if there are complex functions in a file or not. This was the reason that we defined the $M_{ef\%}$ measure to estimate the complexity of a file. This measure can identify complex files irrespective their size.

On one hand if a complex file is modified many times then, according to an earlier observation and two other studies [108, 109], it is most likely a hard-to-maintain file. On the other hand making modifications in a complex file creates likelihood that we did a faulty step in that file. These two considerations and the observation that correlation between NR and $M_{ef\%}$ is low (0.10 / 0.09) motivates us to count the risk as the product of effective complexity and number of revisions.

$$\textit{Relative Risk} = M_{ef\%} * NR \qquad \text{(Eq. 3)}$$

This number indicates the likeliness of a file being defect-prone, difficult-to-maintain or difficult-to-manage. We call this kind of combination superposition of measures as it reflects the joint magnitude of two measures. For example if a file has NR = 20 for one week period and has Effective_M% = 80% at the end of that period we get Relative Risk = 80 * 20 = 1600.

The product of NR and $M_{ef\%}$ does not show how much the absolute risk exposure is, it rather shows the relative risk compared with other files. The product also holds the property of having 0 risk, as the risk is 0 when either NR or $M_{ef\%}$ are 0, indicating no change or no complex function in a file. However there is no upper bound of risk as increasing NR number does not imply linear increase of risk.

Periodically collecting top risky files by this measurement and discussing them within reference group, we established two thresholds, by which we could determine whether a file is considerably risky or not. The thresholds we defined for telecom software were:

1. If the Relative Risk > 1000 for a file in a week development time period then the file is considered highly risky
2. If the Relative Risk > 500 for a file in a week development time period then the file is considered moderately risky.

Figure 6 shows how the $M_{ef\%} * NR = 1000$ and $M_{ef\%} * NR = 500$ hyperbolas separate files with high and moderate risks. By modifying the numbers 1000 and 500 thresholds, we can move the hyperbolas thus including or excluding more files in the list of high risky files. These thresholds were determined the following way:

1. Relative Risk of all files is calculated
2. Files are sorted according to Relative Risk values, so the first file has the greatest Relative Risk number, the second file has the second greatest Relative Risk number, etc.
3. Evaluate top files one by one with designers and determine a number that by designers' perception is a point above which files can be considered risky.

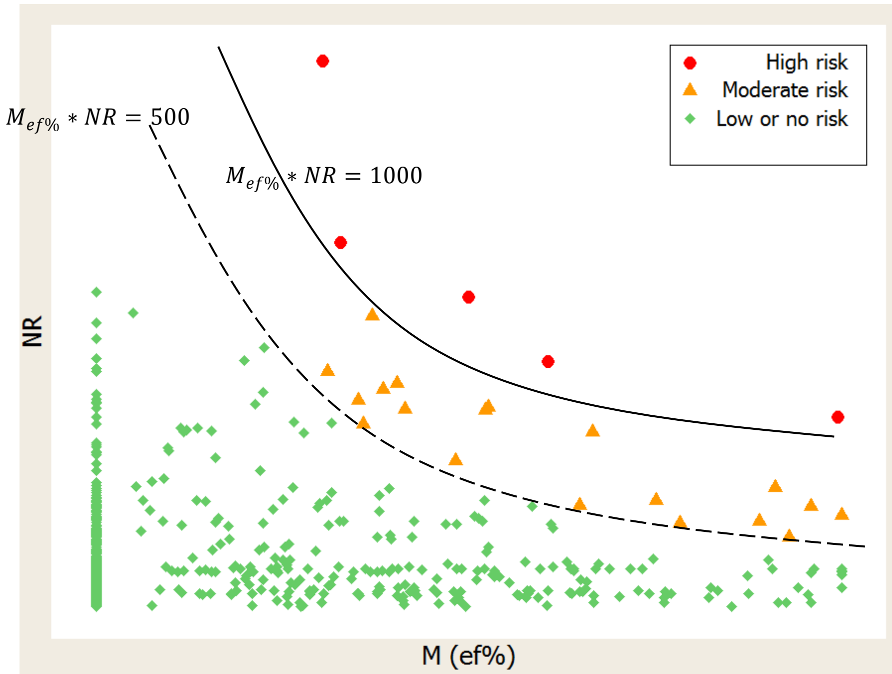


Figure 6 Filtering risky files by defined thresholds

We calibrate the results repeating this process several times. The number of files that the organization can handle in risk mitigation is also considered when establishing a threshold: Too many risky files are not likely to be handled effectively thus bigger threshold values can be established to choose fewer and the most risky files.

The threshold defined for automotive software was different. Most likely the smaller size of the product and the lower number of designers resulted in having in average 3 times less revisions per file than it is for telecom product. However by observing top files over several weeks that had highest Relative Risk value we could establish a threshold to distinguish risky files: If Relative Risk > 200 for a file in a week development time then the file is considered risky.

5 EVALUATION

We carried out two steps of evaluation process:

1. To evaluate the risk of defect-proneness we correlated the number of error reports (ER) with the Relative Risk of files. This activity permitted us to understand if the files having highest Relative Risk number also have the highest number of errors.
2. To evaluate the risk that the files might be difficult-to-maintain or difficult-to-manage we communicated identified files with designers of areas that the file belongs to.

5.1 Correlation with Error Reports

We collected post release ERs per file for 4 service releases. Then we calculated the Relative Risk of files for the time between planning and releasing. The analysis showed that there is a strong correlation between the two measures ≈ 0.70 for all 4 releases, however there were concerns regarding what this correlation shows. The question is how do we know which files are the root causes of ERs and which ones are not? There is no well-established view what ER number per file shows. For example if a simple header file is changed because of ER correction then the change might be caused by changing a complex function in another file, which has been the root cause of the ER. In our correlation analysis we found out several simple files containing ER reports in them. After checking with designers we identified that they are not the root cause of the problem and are changed due to modifications of other files. These simple files "infected" by ERs of other files in the data reduce the correlation coefficient of Relative Risk and ERs. Nonetheless, the correlation analysis was valuable as high correlation between ER and Relative Risk confirms that complex files that are changed often are more fault-prone.

Another concern is that the Relative Risk measure does not distinguish between *big complex* files and *small complex* files, whereas big files are naturally expected to contain more defects proportional to their size. It is worth to notice also that a risky file might not have ER in a particular time period but it might have high likelihood to have ER in the future.

5.2 Evaluation with Designers in Ongoing Projects

At Ericsson we developed and deployed a measurement system for regular usage in the organization. The measurement system was developed based on ISO\IEC 15939 standard and two other studies [87, 105]. The measurement system runs on a daily basis and identifies the files that are risky for the current week development time period. Figure 7 presents a picture of a measurement that visualizes the results: One file with high risk and three files with moderate risk. The designers can follow the link (Source Data) available on the bottom of

the gadget to find all the relevant information regarding risky files. In a period of six weeks we collected and reported (weekly) top risky files to the responsible designers of development areas and obtained their feedback regarding risks.

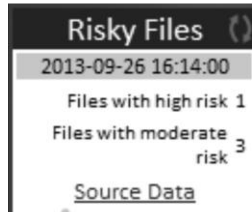


Figure 7 A snapshot of the information product visualizing the risky files

They knew a lot about the details of developed code. The evaluation results at Ericsson were as following:

1. 95% of the identified files were confirmed by the designers to be indeed risky
2. The rest of the files were reported to have low risk
3. Several designers were checking if the file has ER before and deciding risk on that account
4. There were a few files that designers reported to be risky but they were not detected by our measurement system: We detected that our tool failed to calculate the complexity of these files
5. Finally it was difficult for several designers to evaluate the riskiness of a file for a specified week period of time instead of doing it generally. This created an additional difficulty for evaluating the risk for a specified time period

The evaluation at Volvo was done in form of a replication as the product was 10 fold smaller and it was enough with our contact designer to judge if our provided files were risky. The list of files that we determined to be risky was fully confirmed by the designer.

5.3 Impact on Companies

At Ericsson the files indicated by the measurement system were brought up on a design forum where designers discuss plans for improvements. The line manager, who was the direct stakeholder for the measurement system, was provided with a list of files that were constantly appearing to be risky. Refactoring and re-architecting was planned for the risky areas of the product. Early feedback on developed code was of great importance, thus they considered the usage of provided information for release planning also.

Whilst we observed that the refactoring and re-architecting is initiated, at the moment of writing this paper, however, we did not have sufficient quantitative

information to record the effects of usage of the method. For example reduced number of ERs or increased development velocity. By manager's reflection the positive effects were slow but persistent. The organization was moving towards re-architecting and refactoring of risky areas.

Because of the size and not so intense modifications of functionality there was no need of running the measurement system on weekly basis for automotive product. It was rather useful to run the measurement system release-wise to get early feedback on the code to be delivered. At Volvo designers considered to use the measurement system with outsourced development of product when receiving it. The purpose was that when obtaining new code the designers did not know which parts of the delivered code is risky and getting an insight about it in advance was helpful for them to decide where the risky areas are and thus, where improvements' actions must be addressed.

6 RELATED WORK

There have been a few studies proposing methods for identifying the risky elements of software code. Neumann [110] proposes an enhanced neural network technique to identify high-risk software modules. He argues that the combination of neural network and principal component analysis (PCA) can be effectively used for identifying risky software components. Our method can be considered similar with what he proposed, if we consider that the unit step function of neural networks is the product of $M_{ef\%}$ and NR measures with equal weights. Instead of using PCA to remove collinear data dimensions we have chosen pairwise correlation analysis of variables and investigation of their relations, because not only correlation values but also correlation types are important when selecting variables: For example high M number necessitates high Sfout and NCLOC number. Hence it is important to select M out of these three correlated measures. Selim, et al. [111] construct survival models based on size measures and code clones to assess the risk of defectiveness. In their study they defined the risk based on fault-proneness of code, which is important but one aspect of risk. Koru, et al. [112] use Cox modeling [113] to determine the relative risk of software modules' defect proneness. Gondra [114] concludes that Machine learning techniques sometimes are not applicable because of scarcity of the data. Pendharkar [115] supports this idea by claiming that defect prediction models based on probabilistic neural network is not pragmatic and proposes a hybrid approach. Case-based reasoning is proposed by El Emam, et al. [116] to be a good technique for identifying the risk of fault-proneness and difficult-to-maintain classes in the code. Moreover, they validate that the use of different parameters such as different distance measures, standardization techniques, etc. does not make any difference in prediction performance. Our research relies on their study and uses NR as a good predictor of risk. Bakota, et al. [117] constructed a probabilistic software quality model where the internal quality attributes of software code are represented by goodness function for evaluating external quality attributes of software such as analyzability, changeability, test-

ability and stability. Instead of combining the measures in one indicator they use experts' ratings for internal measures to use them as evaluators of external quality attributes. Moreover, they define a rule for representing external quality attributes quantitatively by internal ones. This work is very valuable in terms of new ways of thinking and probabilistic representation of quality. However it is unclear how they chose measures (for example code clones in some cases have positive effect on quality) and why colinearity of measures is not considered. For example they use both cyclomatic complexity and lines of code, but as we show in our work big cyclomatic number necessitates big size. Baggen, et al. [118] also construct an approach for code analysis and quality consulting but again using various size measures. Generally bigger size implies higher maintenance effort, but it is natural as bigger size indicates a bigger product with bigger functionality also which secures higher profit. In fact, what matters is the maintainability per unit of size. In an early phase of our study designers found size measures useless for assessing riskiness hence we had to construct a new approach which calculates the risk disregarding the size, based on $M_{ef\%}$ measure. Shihab, et al. [86] present a large scale industrial study concerned with how code changes can trigger various risks. They use variety of change measures combined with developers' experience. This study is very similar to our study and the two complement each other in many points: Firstly both studies consider the risk to be wider concept then merely fault-proneness of the code. Secondly change measures are considered to be the main source of risk. Even if there is complex code only the change of it can trigger risk. Thirdly both studies are carried out in large software development context and rely on researchers and developers collaboration. The difference between the two is that our study is more focused on combining complexity and change measures in one number in order to develop very simple measurement system. Their study observes the most influential measures among various change measures, bug reports associated with changes, and developers' experience.

7 THREATS TO VALIDITY

The major validity thread is concerned with the evaluation approach with designers. Ideally, complying with well-established statistical techniques, we ought to determine a sample size for number of files for evaluation with designers, randomly chose files and check them on one side with designers and on the other side by measurement system for determining the riskiness. Instead we chose all the risky files found by the system and introduce to designers, as full-scale evaluation with a number of files of sample size was taking enormous amount of effort from organization. Another validity thread is the inconsistency of defining a threshold for risk evaluation. We mentioned earlier the thresholds can vary from company to company. We are certain that files with highest *Relative_Risk* number are the most risky ones but how to measure the risk as an absolute value? This question is still remaining open which could be addressed in the future work. We believe that the parsing capacity of the tools are very

important also as the experience showed that many of the tools could produce severe errors when parsing which are absolutely not acceptable to be used in analysis.

8 CONCLUSIONS

Contemporary software products solve increasingly complex problems leading to increasing complexity of the software products themselves. An uncontrolled growth of complexity over a long time triggers a variety of technical risks that have potential of jeopardizing the business of companies. There is an increasing need for regularly managing these risks, since in Agile software development the self-organized teams deliver small increments of software almost continuously.

This study developed a method and supporting measurement system for identifying the risky source code files and assessing the magnitude of the risk. The method is based on McCabe complexity and number of revisions of source files. The overall results show that out of nine initial measures the superposition of two measures, effective cyclomatic complexity percentage and the number of revisions of a file is a good estimator of risk. The risk is calculated so:

$$Relative_Risk = M_{ef\%} * NR$$

Generally by systematically discussing the intermediate results with reference group we concluded that the complex software code that is changed frequently is risky, hence the superposition of these two measures was evaluated as risk predictor.

The method was evaluated in two projects at two companies, Ericsson and Volvo Group Truck Technology. The evaluation showed that the method is effective in technical risk assessment as well as practical for integrated regular usage within modern software development organizations. The weekly walkthrough with designers showed that it is highly valuable to have a systematic feedback on riskiness of the files. Particularly it is effective to identify the most risky few files out of several thousands, so developers can focus on the most severe risks.

This risk assessment method is developed in Agile context as the studied organizations develop products relying on Agile principles but the method might be successfully used for any type of development. One reason that the results might be different in non-Agile environment is that NR measure might not be as strong indicator of potential problems as it is in Agile environment. In Lean production new features are provided to the customer continuously therefore it was important to have risk assessment based on a measure that reflects the change of code over time. Complex code is not always a problem if it is not changed or maintained. Agile principles in big organization imply that one team can develop code which might be maintained by another team later. If developed code is complex and hard-to-understand, it can become a problem when new teams are

assigned to maintain that code, therefore a fast feedback is crucial on developed code.

The impact of this method is two-fold: In the short term it led to establishing two online, daily updated, measurement systems at the companies. In the long-run it triggered refactoring activities. This unique opportunity to work openly with two companies at the same time led to knowledge sharing between them and learning company-to-company with the researchers as catalysers.

Our efforts are directed towards creating integrated technical risk management methods for modern software development industry. The further work focuses on the extension of here presented method to identify the risky functions/methods in the code. Also, there are plans to expand the study on models for companies working with model-based development.

PAPER 3

A Method for Automated Reviews of Textual Requirements

ABSTRACT

Conducting requirements reviews before the start of software design is one of the central goals in requirements management. Fast and accurate reviews promise to facilitate software development process and mitigate technical risks of late design modifications. In large software development companies, however, it is difficult to conduct reviews as fast as needed, because the number of regularly incoming requirements is typically several thousand. Manually reviewing thousands of requirements is a time-consuming task and disrupts the process of continuous software development. As a consequence, software engineers review requirements in parallel with designing the software, thus partially accepting the technical risks. In this paper we present a measurement-based method for automating requirements reviews in large software development companies. The method, Rendex, is developed in an action research project in a large software development organization and evaluated in four large companies. The evaluation shows that the assessment results of Rendex have 73%-80% agreement with the manual assessment results of software engineers. Succeeding the evaluation, Rendex was integrated with the requirements management environment in two of the collaborating companies and is regularly used for proactive reviews of requirements.

1 INTRODUCTION

A well-defined software requirements specification is a prerequisite for a high quality software design. If a specified requirement is not clearly defined, then software designers consume much unnecessary development time on its clarification. Moreover, if a requirement is not clearly defined, then there is a technical risk that it will be misinterpreted, designed incorrectly, and cause late design modifications [119], [63], [120]. For this reason, software engineers aim to conduct requirements reviews as fast as possible so the design and verification phases can be started with a well-defined set of requirements.

Although there are several studies arguing that early and fast improvements of requirements pay off in software development organizations over time [121], [122], [123] and frameworks for improvements also exist [124], [125], [126], in *large continuous software development projects* manually conducting fast reviews before the start of software design often is not possible because of two main reasons:

1. The number of regularly upgraded or delivered requirements is several thousand [127], so manually reviewing such an amount of requirements takes substantial amount of time
2. Continuous software product development relies on continuous feature delivery to their customers. In such projects most of the development activities run in a continuous manner, therefore it is not an optimal solution to temporarily stop the design and verification activities until the review process is finished

Many researchers have reported that for large software development projects manual reviews of requirements is recognized as a time consuming task [128], [129], [130], [58], [131]. For this reason many software engineers prefer to integrate the review process with the software design and verification, which is believed to be the optimal solution. In other words, software engineers review every requirement just before its implementation, and if there is an unclear specification, they clarify it with the requirements analysts.

As Kauppinen, et al. [122] reports, practitioners find the tool support is a key for efficient reviewing process, as the size of software products grow. Software engineers at the four companies that we collaborated with, also consider that automated means for requirements review would be of great help. Based on their practice, they had a perception that in each of the software development organization, out of thousands of requirements only 3-5% needs improvements. As the collaborating practitioners stated, *"Even if we could locate half of this 3-5% just in time of their delivery, then we could considerably reduce the overall development effort".* Additionally, we learned that requirements analysts prefer getting fast feedback because it becomes easier to remember the requirements they just wrote, and thus the correction becomes easier.

On this backdrop, our research aimed to create a method for automated requirements' reviews and ranking based on needed improvements. The ambition was that the method should be simple to use, straightforward to help improving requirements, and easily integrable in any software development environment. In a pursuit of such a method we addressed the following research question:

> *How can we automatically rank the textual requirements according to their need for improvements in large continuous software development projects?*

Our key contribution is a measurement-based automated method which identifies requirements that need improvements. The more detailed contribution of the paper is as follows:

1. Four measures of internal quality of textual requirements
2. A combined measure (QI_R) for requirements ranking
3. Rendex – a method for requirements ranking based on QI_R
4. Evaluation results of the ranking accuracy and the use of Rendex in four large software development products in four companies

The research was carried out with collaboration of four large software development companies: Saab, Grundfos, Volvo Group, and Volvo Car Group. In the inception, we conducted an action research project in an organization in one of these companies. A reference group of software engineers was formed, which supported us with systematic feedbacks in designing and evaluating the measures. Afterwards, an initial evaluation with six software engineers and 90 requirements in three companies showed that the assessment of Rendex has 73-80% agreement with the manual assessment of software engineers. In the fourth company, where the evaluation was conducted qualitatively, Rendex could locate the needed improvements in the top 5% of the ranked requirements. Consequently, the method was integrated with requirement management systems in two of the companies and used in a systematic basis for proactive requirements' reviews. The other two companies used the method in a semi-automated fashion.

As a final remark, we must say that Rendex is a result of an action research project, and therefore it needs more evaluations so that we can understand the possibilities of its generalizability. With the promising results we also found that Rendex should be calibrated to companies' specific requirements. To provide further guidelines on more generic use of Rendex we plan a follow-up evaluation with a larger data set.

2 Collaborating Software Organizations and Their Requirements

Volvo Group is a Swedish company that develops and manufactures trucks, buses, and construction machines. The company develops ECUs (electronic con-

trol units) for these vehicles and machines. An ECU is a computer which controls a specified functionality for a vehicle automatically. Such functionalities can be, for example, *climate control* or *break control*. Rendex was evaluated in an organization that develops the ECU of chassis for the trucks.

Saab is a Swedish company which develops and manufactures products and services for military defence and civil security. One of the security equipment is a ground radar system, which detects flying objects and reports information about them. The radar system is controlled by a computer which performs the object identification and analysis. Rendex was evaluated in the organization that develops software for the radar system.

Volvo Car Group is a Swedish company that develops and manufactures vehicles in the premium segment. Similar to Volvo Group the company develops ECUs, which perform variety of functionalities for cars. Rendex was evaluated in an organization which develops the *climate control* unit of the cars.

Grundfos is a Danish company that develops and manufactures pumps for variety of purposes. The company also develops software for electronic systems used in pumps. The software system enables the automatic self-control of the pumps' functionality as reaction to changing external parameters, such as liquid pressure, temperature, or volume. Rendex was evaluated in an organization which develops the electronic control unit for water pumps.

The organization in which Rendex was developed, did not want to reveal its name or to which company it belongs to. However, we shall say that it was an organization in one of the four companies, and it is not any of the four organizations where Rendex was evaluated.

The number of requirements in each of the organization was several thousand. The requirements that we analysed in the companies were of two levels of abstraction: *component level* and *subsystem (system) level*. Subsystem level requirements were more general, containing more holistic description of system's functionality. These requirements were written by requirements analysts who interact with customers. These requirements usually were small in size, typically 2-5 lines of natural language text in a page of A4 format. Component level requirements were detailed and comparably large. They were written by requirements analysts who work closely with software designers and testers. These requirements specified how exactly every functionality should be implemented. They were written in natural language but often did not follow grammatical rules and could contain tables, natural language pseudocode, camel-cased words, etc. The size of these requirements could stretch from 1 line to a whole page.

3 INTERNAL QUALITY MEASUREMENT MODEL OF REQUIREMENTS

Following the practice of the four companies, a textual requirement needs improvement if it is *difficult to understand and interpret for implementation and test.* Usually *vague specifications* and *complex descriptions* are the common sources of difficulty of understanding. There are also other sources, such as incorrect or infeasible specifications. However, these sources mainly can be identified by customers or requirements analysts in the phase of requirements elicitation. Thus in the specification phase the main concern with the requirements is whether they are clearly understandable. That is why reviewers mostly care about understandability of requirements for facilitating the implementation and testing. Understandability is affected with several internal properties of requirements [132], grouped under the umbrella of *internal quality.* So in order to quantify internal quality and rank requirements we aimed to develop measures of *internal quality properties*. A well-known set of internal quality properties for software artifacts is proposed by Briand, et al. [16], which we used in this study. An overview of the measures and properties are presented in Figure 1.

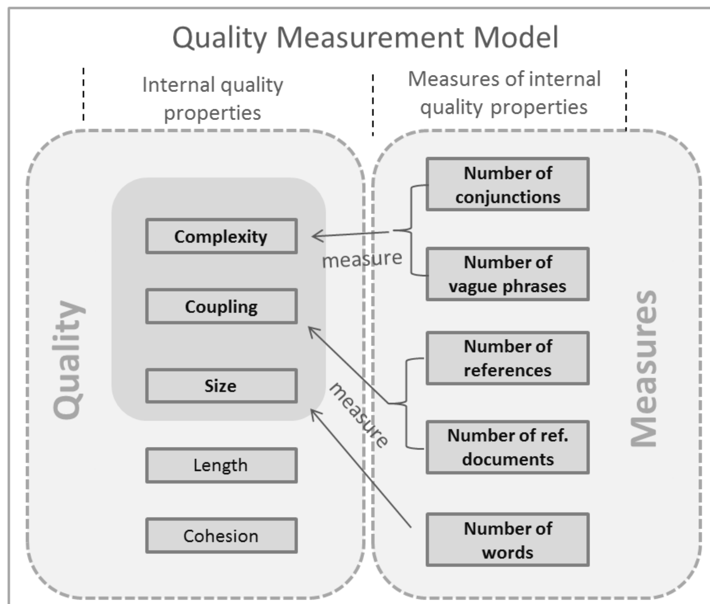


Figure 1 Requirements internal quality measurement model

The left side of the figure represents the internal quality properties of requirements: *complexity, coupling, size, length,* and *cohesion*. The right side of the figure illustrates our designed measures. We could measure aspects of *complexity*, *coupling*, and *size*. The arrows between the measures and the properties indi-

cate the designed measures per property. *Complexity* is generally defined for the concept of system and is closely related with elements and their interconnections in the system. High complexity of the system directly affects its understandability. We use a definition of *complexity* provided by Rechtin and Maier [133] and adapt it to textual requirements the following way: *The complexity of a requirement is defined by its number of different actions so connected or related as to perform a unique function.* As this definition indicates, we are not concerned with the syntactic complexity of a requirement's text. This means we define complexity from the standpoint of system design, hence, this definition is not anyhow concerned with the syntax or morphology of the text. However, we should notice that if there are more actors and relations described in a requirement's text the syntax of that text can become more complex.

Coupling, is generally defined for a subsystem, as to indicate how it influences or is influenced by other elements of the system. We use the concept of coupling adapted from ISO/IEC/IEEE 24765 international standard of Systems and Software Engineering Vocabulary [134] and define it for a requirement the following way: *Coupling for a given requirement indicates the strength of its relationship with other requirements, software modules, variables, or external documents.* This definition indicates that strongly coupled requirement may have many references to software modules, other requirements, or external documentation, which makes a requirement difficult to understand.

Size is a well-known property widely used in software measurement. From quality assessment standpoint, the basic assumption on this property is that if an artifact is too big then it is hard to understand and manage [135]. We used this assumption in our study and designed a size measure for internal quality assessment.

Length is another property that is used in software measurement. Known length measures are *depth of inheritance* or *nesting depth of blocks* in coding. Initially we designed length measures for requirements, however they were evaluated to be ineffective quality indicators (see section 4.6).

Cohesion is the last property in the model. We can refer to two types of *cohesion* of a requirement. One is the linguistic *cohesion*, which indicates how smoothly the members of a sentence are linked in order to deliver the meaning of the sentence to the reader. More discussion on linguistic *cohesion* is provided in subsection 4.6. The second type of *cohesion* refers to the technical representation of a requirement, and can be defined as follows: *Cohesion of a given requirement indicates the manner and the degree to which the actions specified in the requirement are related to one another [134].* As the definition suggests cohesion is an important internal quality attribute for natural language requirements, because it indicates how well the relation of actions is described in a requirement. Two meaningful sentences that specify the same requirement may not have the same cohesion in their description, and therefore, they will not be equally understandable. Unfortunately we could not design a simple measure for the *cohesion* property due to its subjective nature and the difficulty of ex-

tracting semantic information based on syntactic analysis. Generally there are promising techniques for designing cohesion measures, for example Latent Dirichlet Allocation [136], however using sophisticated techniques compromise our aim of creating simple and easily integrable method. We decided therefore to adhere to simple measures.

4 DEFINING THE MEASURES

Table 1 presents the designed measures of three internal quality properties of textual requirements.

Table 1 measures and their measurement method

Name	Abbr.	Definition	Measurement method
Number of conjunctions (complexity)	NC	The number of such linguistic conjunctions that indicate a relation of two related actions in a requirement	Count the overall number of occurrences of the following conjunctions (27 of them): *And, after, although, as long as, before, but, else, if, in order, in case, nor, or, otherwise, once, since, then, though, till, unless, until, when, whenever, where, whereas, wherever, while, yet*
Number of vague phrases (complexity)	NV	The number of such phrases that indicate either imprecise definition or multiple interpretations of a requirement	Count the overall number of occurrences of the following words and phrases (26 of them): *May, could, has to, have to, might, will, should have+past participle, must have+past participle, all the other, all other, based on, some, appropriate, as a, as an, a minimum, up to, adequate, as applicable, be able to, be capable, but not limited to, capability of, capability to, effective, normal*
Number of references (coupling)	NR	The number of variables such as state, sensor data, and module names in a requirement	Count all unique words containing at least one capital letter not in the beginning of the word or at least one underscore in the word. Examples are: *OperatingHours_log, RoomClimate, ReducedLoadMode*
Number of reference documents (coupling)	NRD	The number of references to standards and other documents in a requirement	Count such phrases which indicate reference to documents and standards. In case of one of the companies these are the phrases which indicate references to documents: defined in reference, defined in the reference, specified in reference, specified in the reference, specified by reference, specified by the reference, see reference, see the reference, refer to reference, refer to the reference, further reference, follow reference, follow the reference, see doc.
Number of words (size)	NW	The number of words of a requirement	Count the number of words in the requirement.

The last column of the table presents the method (the complete list of the specified keywords) for measuring a given measure. For example, if we need to calculate NC measure of a requirement we must count the overall number of occurrences of the specified keywords in a requirement. The complete list of the specified keywords for NC (27 keywords) is presented in the second row of the last column of the table. It is important to notice that when measuring any of the following three measures, NC, NV, or NR, we must strictly follow the rules presented in the last column of the Table 1. This is because the list of keywords that we present is selected through a rigorous evaluation process and in case of alteration the measurements will be exposed to inaccuracies. NRD is the only measure which is context dependent and should be calibrated when using for a particular organization.

In the next subsections we present details about each measure and show several examples of how the aforementioned measures affect the requirements. The examples that we present are not complete requirements but complete sentences extracted from requirements, which can help understanding the measures and their use.

4.1 The Number of Conjunctions as a Complexity Measure (NC)

A conjunction is a part of a sentence that connects words, clauses, or sub-sentences. The analysis showed that in textual requirements the majority of all known conjunctions are used to show relations of actions. All such conjunctions are included in Table 1 to define the measurement method for NC measure. There were a few conjunctions which not always showed a relation of two actions, and therefore, they were not included in Table 1. These conjunctions are “than”, “that”, “because” and “so”. Compound conjunctions, such as “even though”, which already contain one of the simple conjunctions listed in the table, are also excluded. Every conjunction indicates a relation. Consider the following simple example extracted from a requirement of subsystem level:

> **In order** to transmit valid data within 100ms SIU shall not respond to external signals

In this requirement there is a relation of two actions connected by the conjunction “in order to”. This conjunction specifies the condition (not responding to external signals) in case of which a given module shall be able to conduct a specified action (to transmit valid data). Here is another example which is extracted from a requirement of component level:

> **When** ContainerType changes to “not valid” then ContainerCapacity should be set to the last value **as long as** VolumeReset is requested.

In this example there are three actions connected by two conjunctions. The first one is used in order to specify how two actions should follow one another. The second one specifies the time interval where the specified relation of the two actions must be valid. When the number of conjunctions increases the number of relations between actions also increases. This leads to a complex description of many actions and relations, which creates difficulty for understanding how to implement and test the given requirement. Particularly, it becomes hard to follow how the actions comply with or has mandate over each other, and what the sequences of actions are in relation to one another.

We also observed that this measure is the most context-independent measure among the test of the measures we defined. Conjunctions are present in any type of requirements specification and show exactly the same phenomenon – relations of actions.

4.2 The Number of Vague Phrases as a Complexity Measure (NV)

As we mentioned earlier the complexity of a requirement is defined by described actions and their relations in the text. While the conjunctions indicate the actions and their relations, the vague phrases indicate the unclearness of the actions or actors, and their relations. In other words vague phrases introduce interpretative nature to the requirement description. Consider the following simple example extracted from a requirement of subsystem level:

> **All the other** transitions than from and to programming session shall not affect the ability to execute non diagnostic tasks

In this example there are two problems with the requirement. First, the requirement starts with a phrase that assumes a previous specification, which we have to explore to understand the context. Second, the expression “all the other” does not specify what other values can be expected. Presumably, this way of writing assumes that there are two sets of values of a given variable and for each of the sets different courses of actions are executed. However, our observations showed that “all other” values usually are not predefined in the document and such a way of writing rather shows that the requirements designer has tried to include all unspecified possible scenarios in one sentence without proper consideration of them. A similar word is “some”, which has the same effect on requirements as the phrase “all the other”. Here is another example from a requirement of component level:

> If the ShortStopHeater_hdlr does not detect open circuit, that **could** be interpreted as thermoswitch is "opened".

In this example the modal verb "could" is used. The first part of this requirement contains a condition which may or may not occur. The second part states that something *could* occur if the first condition occurs. However, the use of "could" does not let the reader know whether the second part should be implemented or it is a consequence that should be considered by the reader. We observed that several modal and auxiliary verbs most of the time introduce such an interpretative nature to the text of requirements. Modal verbs that indicate *wish* or *desire* instead of a requirement that must be developed are ambiguous. Examples are "could" and "might". Auxiliary verbs that indicate past tense or unaccomplished desires are ambiguous. Examples are "must have had" and "should have been". The full list of these phrases is presented in Table 1.

Often, in a requirement there is a need for specifying a range or a measure of "something". This "something" can be speed, frequency, intensity, mode, acceleration, etc. We observed that, there are several qualitative words that in practice are commonly used for such specifications but are inadequate and thus introduce inaccuracy in requirements. Such words are: adequate, effective, efficient, normal, etc. It is important to mention that we can find many more synonymous words to these ones, for example by using a thesaurus dictionary. However, observing many examples of requirements in companies, we found that there are only a small set of words that are often used and are source of ambiguity in the requirements. This is the reason why the list of vague phrases included in Table 1 is not very rich.

We also found many phrases that have tangible likelihood to introduce vagueness but do not necessarily do so. These phrases were omitted in our measurements, as they introduce inaccuracy in the measurements. We widely used existing literature ([137], [138], [139]) for designing NV measure with additional evaluation.

4.3 The Number of References as a Coupling Measure (NR)

This measure is designed to indicate the number of reference items in a requirement that are referred with special names and require certain knowledge about them. These references can be signals, states, modules, or functions which have clearly defined roles. Consider the following example from a requirement of component level:

> If **ContainerCapacity** or **ContainerWarn** or **ContainerTemp** is set to **Not_Valid** then **ContainerAlert** shall be set to **Not_Valid**.

As the example illustrates there is a state (**Not_Valid**) and four variables in the requirement. These variables are defined elsewhere in the requirements specification, appended documents, or source code. To implement this requirement, software designers should have a good understanding of the referenced varia-

bles in it. Many such variable names in a requirement indicate that much knowledge is required to understand how all these variables can and will interact together. We observed that for requirements of component level these references are one of the main constituents of the requirements and by the perception of the software engineers a high number of NR can indicate problems. The NR was measured by counting all the words that contain at least one capital letter or underscore in the middle of the word. This writing convention was well-established in all of the companies for requirements of component level. This was influenced by the principles of software coding, where the names of variables and functions also contain camel cases and underscores. Because of this convention it was easy to measure NR accurately in the four companies.

4.4 The Number of References to External Documents as a Coupling Measure (NRD)

NRD is designed to indicate the references of external documents found in a given requirement. External documents can be international standards, company standards, or other external specification documents. We found out that when designers encounter reference documents in requirements, they have to look through them in order to understand how the implemented requirement should comply or be consistent with the documents. Often, there were no specifics on how exactly the requirement should comply with the referred document. Consider the following example extracted from a requirement of subsystem level:

> The C concept shall allow changes in the configuration of the M modules after the software has been built. For detailed specification of which modules and parameters are changeable **see reference** R configuration specification.

In this example the first sentence states that a specified concept should allow changes in certain modules. The second sentence purports to specify the exact modules which are changeable. The second sentence, however, only refers to a document where the necessary information can be found. Usually the referenced documents contain big amount of other information also, so it is hard to find the necessary information.

In different organizations there are different conventions or habits of how to refer to a document in a requirement. In order to measure NRD in a particular organization, we identified the common phrases for referring to documents in that particular organization. In Table 1 we have specified the phrases, which are specific for counting NRD in one of the companies. In order to define a measurement method for calculating this measure in a particular organization, we recommend using likely keywords that might be used for referring external documents. For example “see document”, “see the standard”, “document”, etc. Making searches based on many synonymous keywords in a large requirements

specification can reveal what keywords are generally used for referring a document. Such an analysis then will allow understanding whether there is a set of keywords that can exhaustively capture all references (or nearly all references).

4.5 The Number of Words as a Size Measure (NW)

Counting the number of words is a commonly used measure for measuring the size of a text. This measure was also used in our analysis in order to understand how it correlates with the newly defined measures and whether it affects the internal quality of requirements.

4.6 Measures Considered but Not Used

We found that *pure grammatical problems* with requirements were not tangible in the practice of the companies. This was true for both high level and low level requirements. The most likely reason is that practitioners who write requirements have reasonably good language skills and actually write reasonably high quality requirements from the syntactic perspective. Even though in low level requirements the natural language syntax is not always assessable as a grammatical construct, the “syntactic” understandability of the flow of the text is still reasonably good. But the understandability of the requirement itself, which has a pivotal role in designing software, can suffer due to high *complexity* and *coupling*. Even if there were small morphological or syntactic problems, such as putting wrong propositions or using the adverbs in wrong positions of a sentence, they hardly ever became an issue for the software designers. We consider that the cohesion analysis of pure morphology or syntax is not desirable in this context for two reasons:

1. There are different styles of writing requirements in every company, and these styles of writing are not necessarily grammatically assessable. Such examples can be tables of specifications, symbolic representations, pseudocode, and briefed clauses. Such requirements cannot be grammatically assessed, because they do not represent “sentences” as linguistic constructs.
2. Following the previous point, the measures which are designed for morphological or syntactic analysis usually follow a set of defined rules enforcing how a requirement should be written, but they do not necessarily reveal the actual problems in requirements

We identified *several phrases* which can influence the requirements’ quality but do not necessarily do so. Examples are: “than”, “that”, “during”, “easy”, “fast”, “passive”, “should” etc. For example the word “that” is not always used as a conjunction, but a pronoun or an adverb, therefore, it does not show interconnections of actions. The keyword “fast” can be used so: “the signal should be as fast as 20 units per second”. Counting these keywords reduces the measurement accuracy and introduces construct validity threat.

The *nesting level* in structured text such as with *bullet points* or *numbering* represents a measure of *length* property. This measure is analogous to the *nesting depth* in software coding, that is the reason we considered evaluating it. However, while in code *nesting* introduces complexity, in natural language text it rather shows how the text is broken down into more simple and comprehensible pieces. In large requirements bullet points were usually used to indicate independent pieces of functionalities which comply with a general condition for all of them. This was an effective tactic for making a large but simple requirement. In many cases we also observed that *bullet points* effectively replaced *conjunctions*. Finally, *nesting level* in requirements was not as deep as its analogous measure in the code. While using *bullet points* in some cases simplify a requirement, in other cases they do not have any tangible effect, so *nesting* in text that is based on bullet points was excluded it from the final list of measures. Another measure of length property is the *hierarchical level* of a requirement among other requirements. However we found that the *hierarchy level* of a requirement has no tangible impact on its quality. .

Three *measures of evolution*, *number of revisions, number of versions* and *number of variants* of requirements were initially measured. Unlike their analogous measures for source code [140], these measures did not have significantly high values, usually limited in a narrow interval of [0, 4]. After the evaluation they were shown to be poor indicators of internal quality and were not considered further in this research. Initially, in order to detect the *number of actions* in the requirements, we measured the *number of verbs (imperatives)* and *punctuations*. However the *number of verbs* was not an accurate measure due to the following reasons:

1. The number of verbs in a sentence does not always correspond to the number of actions specified in that sentence
2. Many requirements were written in natural language text but did not represent sentences as grammatical constructs, and therefore often omitted the use of verbs. For example: “If signal A or B then default state activation”

Punctuations (comas, colons, semicolons) indeed connect actions, but are not always used to show such connections. This is because several *conjunctions*, which do not require separation with comma, can connect *actions* as well, thus the *number of punctuations* do not approximate the *number of relations* in a given requirement.

4.7 Range of Measurement Values

It is important to notice that the measures had significantly different values from each other in the measured set of requirements. Of all the measures NC had the biggest values (excluding the size measure). For the requirements of component level, in average NC had nearly twice as big value as NR. In average, it also had about six times as big value as NV. The NRD was infrequent. There

was a little chance that a requirement can have NRD = 1 value. For the requirements of subsystem level, the proportions were nearly the same with one exception: NR was more infrequent. This can be explained by the fact that high-level requirements are much closer to daily language and thus do not contain many special names of variables or functions.

5 RESEARCH DESIGN

The research was initiated in a large software development organization as an action research project with collaboration of the authors and software engineers. During a period of nine months we developed Rendex in the collaborating organization. Later we conducted four evaluative studies in the four companies: in three of them Rendex was evaluated quantitatively and in one of them qualitatively. In the next subsection we describe the process of developing the measures.

5.1 Action Research for Designing Measures

To develop the measures we established a collaboration unit with the researchers (the authors of this paper) and a reference group of software engineers in the collaborating organization. The principles of action research were used Susman and Evered [42], Baskerville [141]. The reference group consisted of one manager, one test leader, one software designer, and one design architect. The software engineers had 15 plus years of experience in software development and were regarded as the key engineers of the organization. The researchers established monthly formal meetings with the reference group, as well as many non-formal meetings.

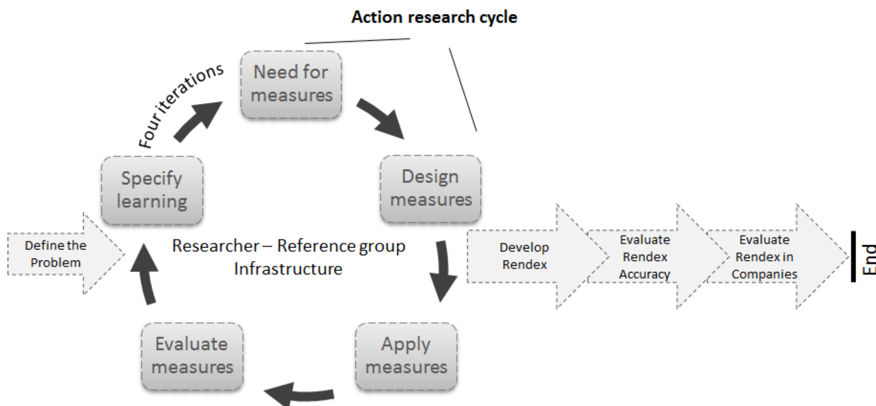


Figure 2 An overview of the research method

Figure 2 presents the action research cycle and three consecutive research activities. The cycle with four iterations was carried out for designing the measures. Then Rendex was developed, and its ranking accuracy and use in the companies were evaluated. The next four subsections describe the process of designing the measures through the action research cycle.

5.1.1 Access to the data

The data were extracted from the requirements management tool into a RIF (Requirements Interchange Format) document. The document contained all the necessary information about the requirements, namely, requirements names, identification numbers (ID), revision numbers, variants, and descriptions.

5.1.1 Design measures

In order to get started with defining measures, the researchers and the engineers of the reference group met and proposed an initial set of measurable aspects of requirements supported by certain rationale. To propose an initial set of measures in the very first iteration, the researchers requested from the reference group to identify and present two sets of requirements: The first set should contain 10 requirements that need improvements by the engineers' perception. The second set should contain 10 requirements that do not need any improvements. The software engineers organized a workshop, where with other six engineers of their team they discussed and selected the requested 20 requirements and provided to the researchers. In this context *a requirement is regarded as "needing improvement" if software designers or testers find it difficult to understand for implementing or testing.* The researchers examined the two sets of requirements and tried to identify the properties that distinguish these two sets of requirements. The researchers explored linguistic and technical aspects of the requirements and investigated what is measurable. Based on this analysis the researchers designed the initial proposals of measures.

5.1.2 Apply measures

Once the initial proposals of measures were defined the researchers developed a tool and conducted measurements. A Python script was used to parse the RIF document. After parsing, the results were stored in an output file, containing the requirements' name and results of the measurement per requirement.

5.1.3 Evaluate measures

Four evaluative iterations were conducted with the reference group through the action research cycle. Each of the iteration was a workshop with the researchers and reference group, where all the proposed measures were discussed. In the workshops the researchers presented the design of the measures and motivations behind the design. Several requirements with both high and low measurement values were discussed for all the measures. The engineers discussed what the measures mean for them, what they perceive the measures indicate,

and then they proposed suggestions. Considering the engineers' reflections the researchers selected two sets of requirements for continuing analysis and preparing the next workshop session. Each of the set contained 5-6 requirements and was selected based on the following criteria:

1. The first set represents requirements that are difficult to understand but the measures that were designed in the previous iteration could not detect these requirements
2. The second set represents requirements that are easy to understand but they have high values for the measures designed in the previous iteration.

By scrutinizing these requirements, the researchers could understand the reasons of why the defined measures were not adequate for the separated sets of requirements. This preparation allowed the researchers to change or solidify the rationale that was provided for designing the measures and facilitated the process of understanding the generalizability of the measures. Several measures that were considered candidate measures in the first iteration were invalidated and ruled out during the later iterations. Overall about 60 requirements were discussed for designing the measures including the first 20 requirements described earlier in this subsection.

5.2 Developing Rendex

Once the measures were designed, we decided to combine them into a single indicator that can be used for requirements assessment and ranking. However, before combining, we ought to check whether or not there are strongly correlated measures, because strongly correlated measures are alternative measures of the same property. Simply stated, strongly correlated measures indicate exactly the same problem in a requirement. Using both of them for obtaining a single quality indicator can artificially diminish the importance of the rest of the measures in calculating a quality index for requirements [72]. We used Pearson correlation coefficients and correlation plots for correlation analyses. The results of the correlation analyses between the measures are provided in section 6.

To obtain a single quality indicator we conducted regression analysis, because it allows quantitatively determining optimal coefficients of the measures. In order to conduct regression analysis we used sample sets of requirements from the companies which were manually assessed by software engineers in the companies.

Before conducting regression analysis we postulated a formula for requirements ranking, which is similar to a regression equation but the weights of the measures are defined with a qualitative approach, that is, by asking the experts of reference group and summarizing their responses. If the postulated formula performed well in requirements ranking and could be a good approximation for all regression equations (an equation per company), then we could use it as a

generalized formula for requirements ranking. To determine the weights of the measures, we organized a workshop with the reference group and discussed the measures and their influence on the requirements. On several examples, the engineers expressed their understanding of how much the measurement values influence the understandability of the requirements. Summarizing we determined the approximate coefficients of the measures and postulated a formula of quality index:

$$QI_R = NC + NV + NR + 5NRD \quad (1)$$

5.3 Evaluating the Ranking Accuracy of Rendex

This subsection presents the two approaches by which we evaluated the ranking accuracy of Rendex. Both approaches rely on manual assessments of software engineers QI_E, where QI_E is the averaged rank for a requirement's internal quality given by software engineers. In the first approach we used the postulated formula (1) for calculating QI_R of requirements and evaluating how much it agrees with QI_E. In the second approach we employed regression analyses to determine how well the regression equations can predict QI_E.

5.3.1 The first approach: evaluating QI_R against QI_E

In order to compare the results of the automated ranks (QI_R) with the manual assessments (QI_E) we defined the following stepwise process:

1. Rank all requirements using QI_R.
2. Randomly select 15 requirements from the top 100 requirements with highest QI_R (requirements needing improvements)
3. Randomly select 15 requirements from all the requirements that are not included in the top 300 requirements with highest QI_R (satisfactory requirements). The next 200 requirements after the top 100 requirements were omitted intentionally in order to assure that there is a significant measurement discrepancy between requirements that need improvements and requirements that do not need any improvements according to QI_R
4. Create a random mix of the two groups of requirements and get a set of 30 requirements
5. Select two software engineers for assessing the quality of the 30 requirements
6. Ask the two engineers to rank each requirement using 5 values of Likert scale, where 1 means a requirement is absolutely easy to understand and 5 means the requirement needs improvement most urgently. This five-scale of ranking is chosen because there shall be enough ranks to observe the disagreement of ranking between engineers themselves. When conducting the evaluation with engineers we also provided textual description per rank, indicating what exactly the rank means (Table 2).

Table 2 Descriptions of ranks

Rank	Description
1	It is absolutely easy to understand this requirement
2	It is rather easy to understand this requirement
3	This requirement can be improved to make it easy to understand
4	This requirement should be improved, because it is hard to understand
5	This requirement must be improved, it is not possible to understand

7. Combine and average the assessment results of engineers as a final quality index of engineers (QI_E).
8. Classify the requirements into two categories – satisfactory and needs improvement – based on QI_E two alternative ways
9. Alternative 1: Strict case
    a. If the $QI_E < 3$ then the requirement is considered as satisfactory
    b. If the $QI_E \geq 3$ then the requirement is considered as needs improvement
10. Alternative 2: Not-strict case
    c. If the $QI_E \leq 3$ then the requirement is considered as satisfactory
    d. If the $QI_E > 3$ then the requirement is considered as needs improvement
11. Develop a confusion matrix for both strict and not-strict cases by the rules specified in Table 3. The TN, FN, FP, and TP are true negative, false negative, false positive and true positive values correspondingly.

Table 3 Confusion matrix and evaluation rules

	QI_R satisfactory	QI_R needs improvement	Evaluation method
Strict case: $QI_E < 3$ Not-strict case: $QI_E \leq 3$	TN	FP	PA%=100*(TP+TN)/n%
Strict case: $QI_E \geq 3$ Not-strict case: $QI_E > 3$	FN	TP	

12. Calculate Percentage Agreement (PA) for both strict and not-strict cases by the following formula: PA=100*(TP+TN)/n% to assess the agreement of QI_R and QI_E. The number n=30 indicates the overall number of requirements in the sample size.

Since the portion of the requirements that need improvements was much smaller compared to the overall number of requirements (3-5%), a completely random selection of the sample size would result in very few or no requirements turning out to need improvements among the 30 requirements. This fact would

jeopardize the evaluation results. Instead we selected the sample size based on the QI_R, which postulates that 15 requirements out of 30 need improvements and the other 15 are fine. This tactic balanced the two groups of the requirements in the sample size.

The two alternative cases of evaluation (strict and not-strict) are carried out due to a problem of creating binary categorical data: If there are many requirements that get marginal average QI_E number (e.g. average $QI_E = 3$) the estimated agreement (PA) between QI_R and QI_E will be an underestimation or overestimation. By providing two alternative ways of assessments, we can have more objective view on the assessed agreement.

We must notice that the manual rankings of requirements in the sample sets are based on subjective judgments of software engineers. For this reason we tested the congruence level of the two engineers' assessments. Kendall's tau coefficient [142] was used for this test, because the data is of ordinal type and for two raters.

5.3.2 The second approach: conducting regression analyses for obtaining QI_R

Using the same data samples of the manual assessments described in the previous subsection, we conducted regression analyses to determine the coefficients of the measures in the regression equations. Through these analyses we also tested the statistical power of regression equations in predicting the manually provided ranks.

5.3.3 Establishing the evaluation setup in the companies

In each of the companies we chose a software development organization where the method could be evaluated. In each of the organization we collaborated with software engineers, who helped us in accessing their requirements, discussing the adequateness of our measures for their requirements, tuning the measures, and selecting two software engineers for manual assessments of the sample sets of requirements. The collaborating engineers had core knowledge about their requirements management processes and had many years of experience (> 8 years) in requirements management. The engineers who provided the manual assessments were software designers and testers with 10 plus years of experience. They were neither a part of designing the measures nor knew about the measures. The companies preferred not to reveal the exact number of requirements, but we can say that the number of requirements in each organization ranged from 2000-8000. Since the companies did not want to present results mapped on companies henceforth we refer the companies as company X, Y, Z, and Q.

5.4 Evaluating Rendex in Companies

After finalizing the development of the method, all four companies expressed their interest to use the method and find out whether they would like to adopt it for regular organizational use. Due to a variety of factors, the adoption of the tool was different from company to company. We were directly involved in the process of integrating Rendex in two of the companies, which used the same requirements management tool. The third company preferred integrating the four measures in the set of rules that they had defined for writing requirements. The last company used our Python script for semi-automated evaluation of requirements whenever needed. After nearly 6 months of using the method we had two formal meetings with technical leaders of the requirements management teams in the first two companies. In both cases we asked questions about how they think the method performs, whether they feel that its application eases the review process and spares time for them, whether they perceive the tool is accurate in evaluating requirements, and whether the way it is integrated to the requirements management system is adequate for use.

6 RESULTS OF CORRELATION ANALYSES AND SELECTION OF MEASURES

Pearson correlation coefficients between the measures for three companies are presented in Table 4 (the fourth company is not included in this analysis). The analysis shows strong correlation between NW and NC measures for the three companies. There are also significant correlations between NR and NC in case of company X (0,578), between NV and NC in case of company Z (0,625), and between NV and NW in case of company Z (0,563). The rest of the correlation coefficients are weak or insignificant. The absence of correlation coefficients for NRD measure in case of company Y is explained by the fact that requirements of company Y are of subsystem level which do not have high enough values for NRD in order to permit correlation analysis.

Table 4 Correlation analyses results of measures for three companies

X/Y/Z	NW	NC	NV	NR
NC	**0,80**/0,62/ 0,86			
NV	**0,48**/0,23/ 0,56	**0,36**/0,21/0,62		
NR	**0,46**/0,46/0,39	**0,57**/0,35/0,43	**0,14**/0,15/0,29	
NRD	**0,25** / - / 0,29	**0,26**/ - /0,27	**0,04** / - /0,12	**0,14**/ - /0,05

As there is a fairly strong correlation between NC and NW we should choose only one of them. In order to choose a single measure out of NC and NW we analyze their correlation plots. In Figure 3 we can see correlation plots of NC and NW where every dot represents a requirement.

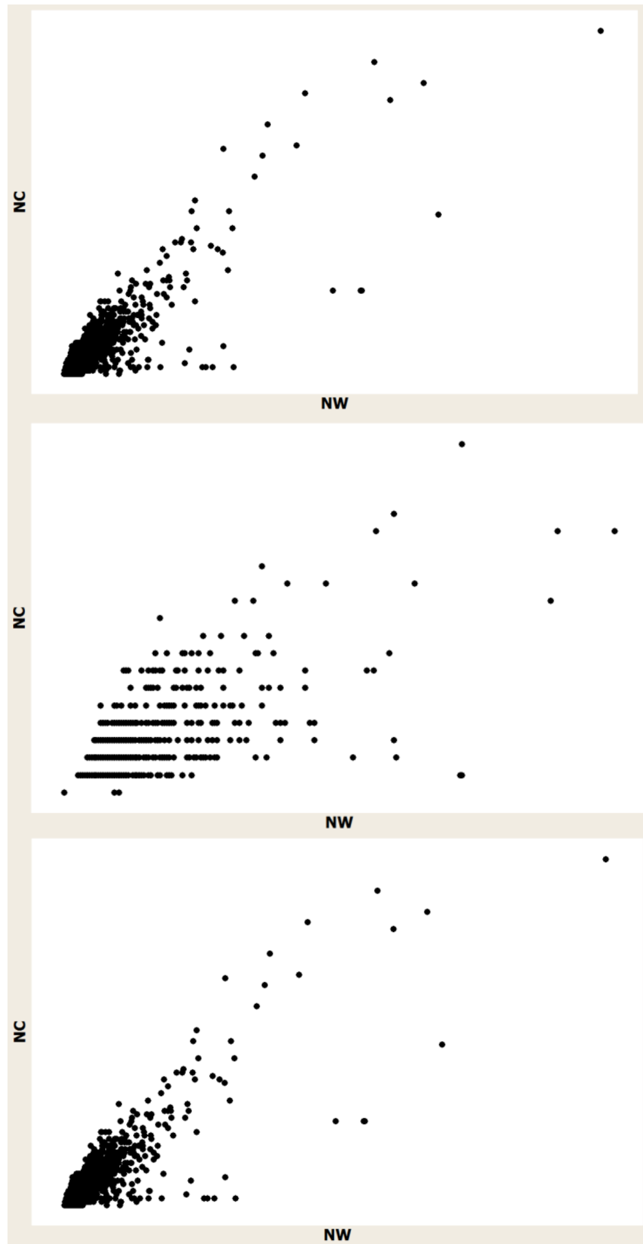


Figure 3 Correlation plots of NC and NW measures for all three companies

These three plots have one general feature: In all of them the dots are scattered over NC and NW dimensions in such a way that *the upper left side* of the plot (along with the axis of NC measure) is empty. This means that there are no requirements that have many conjunctions and are small in size at the same time,

and that is intuitive. Oppositely, we see that there are many requirements along with the axis of NW measure and in the nearby area. This means that there are requirements which have big size but do not contain many conjunctions. Concluding the above two statements we can say that a requirement text with many conjunctions necessarily has many words as well, whereas, a requirement text with many words does not necessarily have many conjunctions. In other words, only NC measure indicates both size and complexity, while NW indicates only size. For this reason we chose only NC to use in the regression analysis.

7 COMBINING SELECTED MEASURES

A weighted sum of the four selected measures is used as a combined internal quality index. The weights of NC, NV, and NR were considered equal by the reference group engineers. This means that adding a conjunction or a vague phrase or a reference to a requirement decreases that requirement's understandability equally. The influence of NRD was considered to be about five times stronger on internal quality. This is reasoned the following way: Having a reference to a document requires considerable amount of time to find the reference and the necessary information in it for understanding how a requirement should comply with it. Similarly, five conjunctions make a requirement so difficult that a designer usually needs a clarification by communicating with a relevant person to understand it. So, considering the weights of the measures we calculated QI_R for a requirement by formula (1):

$$QI_R = NC + NV + NR + 5NRD \qquad (1)$$

QI_R is not an absolute but a relative measure of internal quality, indicating that the requirements with smaller QI_R have better quality. QI_R does not provide a threshold by which a requirement can be regarded either "satisfactory" or "needs improvement". Despite not establishing an absolute threshold, with agreement of the reference group designers we concluded that usually it is preferable that for a requirement $QI_R < 5$ is a good threshold, because in practice the requirements with bigger QI_R values were becoming hard to understand.

8 EVALUATION RESULTS OF RENDEX

Subsection 8.1 presents the PA values for the three companies. It provides correlation analyses results between *the measures,* QI_R and QI_E. In the same subsection the results of agreement between manual assessors are also provided. Subsection 8.2 provides the results of regression analysis and discusses the differences across the companies. Subsection 8.3 generalizes the regression analyses results.

8.1 Results of Evaluating QI_R against QI_E

PA values for the three companies are presented in Table 5. The best results of agreement was obtained for company X, followed by company Z. Both of the companies had requirements of component level, and there is no substantial difference between the results of **strict** and **not-strict** cases for them.

Table 5 Evaluation

Strict / Not strict	X	Y	Z
PA	**80%** / 87%	**73%** / 63%	**74%** / 70%

The somewhat bigger difference of results in **strict** and **not strict** cases for company Y is due to the fact that there were relatively more marginal ratings in the manual assessments, i.e. some requirements got rank 3 which is a marginal value for calculating PA.

In company Q, when we integrated the method with their requirements management system, they had already manually detected about 20 requirements for urgent improvements, but they were still in the beginning of the review process for that particular project. By running the tool we found that all these requirements were among the top 100 requirements that needed improvements provided by the tool. We would like to emphasize that the tool found all these requirements in about two thousand requirements.

Table 6 presents Pearson (R) and Spearman (S) correlation coefficients between the measures and QI_E for three companies. Strong values of the coefficients are boldfaced (R, S > 0,6). Generally the NC measure is strongly correlated with the manual assessments. The rest of the measures have significant but not strong correlation with QI_E.

Table 6 Correlation coefficients for three companies

	QI_E (X)	QI_E (Y)	QI_E (Z)
NC	**0,74 / 0,77**	0,54 / 0,54	0,58 / **0,65**
NV	0,27 / 0,30	0,20 / 0,23	0,52 / 0,57
NR	0,56 / **0,68**	0,43 / 0,24	0,46 / 0,44
NRD	-	-	-
QI_R	**0,72 / 0,77**	**0,62 / 0,60**	0,57 / **0,62**
p value for QI_R	< 0,001	<0,001/ 0,001	0,001 / 0,005

Since the values of NRD measure was very low, we could not conduct correlation analysis. The last row of Table 6 shows correlation coefficients between QI_E and QI_R. The coefficients are strong, and the low p-values show that the likelihood of getting these results driven by chance is too small.

Table 7 presents the tau coefficients of agreement for every pair of software engineers per company. Generally we can see that there is a significant agreement between the assessors.

Table 7 Correlation results between engineers' assessments

Company	X	Y	Z
Kendall's tau	0,56	0,61	0,75
P value	0,001	< 0,001	< 0,001

8.2 Results of Regression Analyses

Regression analyses were conducted to evaluate how well regression equations can predict QI_E . The equations (2) represent the regression equations for the three companies. The variable NRD is not included in the regression equation for company Y, because NRD had too low values for the requirements of subsystem level.

$$X{:}\ QI_E = 4{,}25 + 0{,}076\ NC + 0{,}11\ NV + 0{,}017\ NR + 0{,}27\ NRD$$

$$Y{:}\ QI_E = 1{,}40 + 0{,}093\ NC + 0{,}08\ NV + 0{,}076\ NR \qquad (2)$$

$$Z{:}\ QI_E = 7{,}09 + 0{,}157\ NC + 0{,}029\ NV + 0{,}050\ NR + 2{,}86\ NRD$$

Since QI_E values are of interval scale, the absolute values of QI_E are not meaningful themselves. Therefore, we can simplify (2) equations by multiplying their right side by 100 and removing the constants. Then we can get the following set of equations (3), which is equivalent to equations (2):

$$X{:}\ QI_E = 7{,}6\ NC + 11\ NV + 1{,}7\ NR + 27\ NRD$$

$$Y{:}\ QI_E = 9{,}3\ NC + 8\ NV + 7{,}6\ NR \qquad (3)$$

$$Z{:}\ {:}\ QI_E = 15{,}7\ NC + 2{,}9\ NV + 5\ NR + 286\ NRD$$

Observing equations (3) we can see that the coefficients of the measures do not have any fixed predominance over each other across the equations. This means that we cannot obtain a general regression equation for all three companies in an explicit manner. This observation seems to be intuitive, because the requirements were of different types and the sample sizes were not big enough for finding similar patterns across the companies. Only NRD has significantly higher coefficients in the regression equations, which was an expected outcome, because it has very small values for requirements and higher impact on internal quality by qualitative reasoning of the software engineers. The upper part of Table 8 presents p values of the four measures in the three regression equations corresponding to the three companies. Smaller p values indicate bigger statistical significance in the equations. Small P-values for NC measure show statistical significance for the companies X and Y (0,008 and 0,010 < 0,05), which indicates that NC measure can be used alone as well for predicting QI_E.

In case of company Z the p-values did not gain any statistical significance. The lower part of Table 8 presents R-squared, R, and p values for the regression equations. The values of R-sq. are ranging from 39% to 56%, indicating that 39-

56% of data variation in QI_E can be explained by regression equations. For all three regression equations the p values gain statistical significance indicating that the probability that our results are by chance is very small.

Table 8 P values for the measures and for regression

	QI_E (X)	QI_E (Y)	QI_E (Z)
P values of measures in regression equations			
NC	**0,008**	**0,010**	0,441
NV	0,650	0,766	0,967
NR	0,339	0,050	0,838
NRD	0,404	-	0,148
P and R-sq values for regression equations			
R-sq	56,6%	39,1%	40,4%
R	0,75	0,62	0,64
P value	< 0,001	0,004	0,009

8.3 Generalizing the Results

Let us compare the Pearson correlation coefficients of QI_E *and regression equations* in Table 8 with that of QI_R *and* QI_E that is obtained from applying formula (1) in Table 6.

Table 9 A comparison of correlation coefficients

	QI_E (X)	QI_E (Y)	QI_E (Z)
Regression equations	0,75	0,62	0,64
QI_R from formula (1)	0,72	0,62	0,57
Difference	0,03	0	0,07

This comparison is illustrated in Table 9. The correlation coefficients for the postulated equation (1) are only slightly weaker than the correlation coefficients obtained by regression analyses. The differences between coefficients are presented in the third row of the table. The weakest approximation is found for company Z (difference of coefficients is 0.07). However, even in the weakest case, the difference between the correlation coefficients is not significant. This means that formula (1) can be regarded as a good general approximation of regression equations and thus can be a substitute of them for the all three companies. Saying that, we still find the generalizability is a problem, because coefficients of measures in all three regression equations are irregularly different, and formula (1) is rather an informal generalization.

9 REQUIREMENTS QUALITY INDEX APPLIED IN THE COMPANIES

Summarizing the evaluation results we shall state that QI_R obtained by formula (1) has 73-80% agreement with human assessors of requirements. Alternative evaluation results by the rank (Spearman) correlation coefficients between the automated and manual ranks are 0.60 – 0.77. Since the results show substantial agreement between the manual and automated ranks we consider that the automated ranking can be effectively applied in practice. Hence, in this section we delineate a method for automated ranking of textual requirements. The following steps shall be done in order to rank requirements based on QI_R:

1. Extract all textual requirements in a structured file. These can be .xml, .txt, .rif. The extracted file should contain the names and the texts of all requirements.
2. Calculate the NC, NV, and NR measures for each requirement by the rules specified in Table 1
3. Calculate NRD measure by investigating the naming conventions and referencing conventions in the organization where the measurements are done. Identifying these conventions with fellow designers and requirements analysts is recommended. For NRD calculation consider the help of the measurement rules presented in Table 1 also
4. Calculate the QI_R per requirement based on formula (1): $QI_R = NC + NV + NR + 5NRD$
5. Rank all the requirements by QI_R in descending order
6. Decide upon a cut-off point in the ranked list, above which all requirements shall be manually reviewed and improved. In the practice of the collaborating companies the cut-off point is usually chosen in a way that it separates 3-5% of all requirements with the highest QI_R values.

The collaborating companies found that Rendex has good enough accuracy for automated requirements reviews, because it permits finding most of the requirements that need improvements in only 5% of the overall requirements. In two of the companies Rendex was integrated with the requirements management tools (RMT), so every requirement analyst could get an instantaneous feedback “just in time” of writing requirements. Every requirement in RMT had a new field with a QI_R value. This means that every engineer could choose to see QI_R values of requirements. Figure 4 presents a screenshot of requirements and their quality indices in RMT at one of the companies. The right side of the figure presents several requirements and their quality indices for a given software component. We recommended that the higher quality indices shall have darker color so they can be easily identifiably. Generally the practitioners perceived that the use of Rendex accelerates their working process substantially and allows proactive management of requirements’ quality.

Name	Status	Version	Next V...
§g DesignLevelRequire...	Work	(1)	
TSR_Satisfy	Work	(1)	
VehicleSpeed_ctrl	Work	(1)	
EngineSpeedCo...	Work	(1)	
SpeedControlM...	Work	(1)	
CruiseControlM...	Work	(1)	
EngineManager...	Work	(1)	
vehicleIndicatio...	Work	(1)	

Name	Quality index
TSRRequirementPlaceh	0
EscEngineControlMana	6
EscDowngradedModesI	2
EscAutoNeutralManage	0
BehaviorDescription_Es	4
BehaviorDescription_Es	7
Routine_Control_Calibr	3
Calculate_and_distribut	3
Initiate_synchronize_an	1

Figure 4 Rendex integrated with RMT

One of the technical leaders of requirements management states that: *"If we continue using this method in our organization, I think we are going to spare hundreds of hours till the end of the product release".*

10 THREATS TO VALIDITY

Validity issues of this study are identified and discussed here based on the recommendations of Baskerville and Wood-Harper [143] and Checkland and Holwell [38].

We believe the biggest issue of this study is the external validity. The complexity of the research environment and the multidimensionality of the internal quality of textual requirements hindered us to conduct a study that would satisfy the classical validity criteria of positivistic scientific studies. Particularly, generalizability of the results needs further investigation. Larger sample size of requirements and more companies could most likely help to overcome this issue, but we must notice that conducting action research with many companies and larger sample sizes is not realistic in the scope of one study. We consider the biggest value of this study is that it provides simple and effective measures of textual requirements that are validated in real contexts. The generalization of the regression equations also provides a practical tool for industrial application. Unfortunately the sample size of the requirements was not big enough for obtaining conclusive results for generalization, however, we still found that a simple general method can already be constructed for practical purposes. We are hopeful that in the future additional evaluation of these measures can be conducted, so the possibilities of generalization can be better understood.

The context of the research is confined with requirements of large companies, who have well-established requirements management process. The measures that we developed are only tested on these requirements. It is very likely that small or inexperienced companies can have completely different types of problems which affect the understandability of requirements. Particularly, complexity may not be a problem for them, while the lack of well-established writing conventions can become a large problem. Therefore, the results of this research are most likely more useful for requirements of large products.

While the usefulness of the measures was qualified in practice, the interpretive nature of the analysis did not permit us to estimate the empirical effectiveness of measures in larger scales. This is not a particular problem for this study but rather an inherent problem for complex systems' research [44], as the effectiveness of applying measures in complex systems are hard to quantify. The involvement of several software products in the study permits us partly overcome that problem by observing how well the same measure performs from company to company. Notwithstanding, more detailed examination of these measures would be valuable.

Evaluating the measurement methods for the measures in more contexts would benefit the measures and their use. Particularly the measurement methods for NR and NRD can exhibit problems in practice because these methods heavily rely on the writing conventions of the practitioners. The evaluation showed that the current measurement method for NR is generic at least for the four companies, while the measurement method for NRD should be specific from company to company. This implies that the measurement method for NR should be investigated further to understand how generic it is in different companies and contexts of requirements.

There is also a construct validity threat for formula (1), because the weights of the measures were decided in a qualitative manner. In a way, we considered this formula as more of a priori postulate for an evaluation, and if it turned out to work well, then we could simply use it. To find out whether the formula is a good approximation for all three companies we conducted regression analyses and compared the results. However, when it came to regression analyses results and possibilities of deriving a general equation for ranking, it turned out the weights of measures in the three formulas are different. The problem was that there were no straightforward solutions of deriving generalized weights. We intendedly did not focus on the regression equations per case because the coefficients of the measures could not be effectively explained. Bigger sample sizes of requirements would have been helpful for such an explanation. They could even give a possibility to ultimately decide whether one general formula for assessment is possible to derive. But we leave this problem to be addressed in our future research.

The next construct validity threat is concerned with calculating PA values when there are different scales of data. The assessment values for engineers is limited by [1, 5] integer numbers, while the assessment of Rendex was in an unlimited $[0, \infty)$ interval. In order to compare these two assessments we had to bring them into a comparable interval. One simple way to do so is to derive binary ordinal measurements from their interval measurements, that is: a requirement is either "satisfactory" or "needs improvement". Categorizing data in a binary form creates possibility of evaluation by confusion matrix. However it also brings a new validity threat that is the issue of marginal requirements. If a requirement actually had an average quality, it is "forced" to be either a satisfactory or dissatisfactory one. We could partly solve this issue by additionally creating so called **strict** and **not strict** evaluations, which provide two alternative

overviews on how Rendex performs when comparing it with engineers' assessments. As a final point, we emphasize the fact that the research was validated with four software development organizations, so the results seem to have a good chance to be generalized for textual requirements. Therefore we encourage the use of the measures (method) in software development organizations and appreciate the feedback we might get.

11 Related Work

There are different views in the literature on what quality properties are reasonable to measure, for example models by Fabbrini, et al. [144] or Bøegh [145]. One of the early studies that attempted to define quality measures for textual requirements was conducted by Davis, et al. [146]. Their paper proposes 24 quality properties that can be measured for a requirement, and measures for 18 of them. As the study was one of the first studies in the area of requirements measurement, it is rather a suggestion of possible quality attributes and measures, which could be maintained and advanced. Similarly Costello and Liu [147] proposed software quality properties for measurements, among which there were properties of requirements such as traceability, consistency, volatility etc. Later a study conducted by Hyatt and Rosenberg [138] proposed simple measures for identifying ambiguous phrases in textual requirements and actually used them in a software metrics program.

Since then there have been several studies which conducted research towards creating measures for textual requirements. For example Vlas and Robinson [148] created measures for requirements classification. Based on certain patterns identified in requirements, they could automatically classify a requirement as holding a particular property or belonging to a certain type of requirements group. Huertas and Juárez-Ramírez [149] presented an automated tool for analysing such properties as ambiguity and atomicity of a textual requirement. Kasser, et al. [150] defined measures for assessing the quality of requirements. They determine the words that most likely introduce uncertainty to the requirements, then they calculate those words. Most of the words in their list are evaluated and included in our list. We perceive that the aforementioned studies would benefit substantially from industrial evaluation. This is because the practicality and precision of the designed measures are identifiable only in the organization where the actual requirements review process is conducted.

Gleich, et al. [139] examined the ambiguity patterns in textual requirements and proposed a tool for ambiguity detection. The study is notable by the fact that the ambiguity patterns are explained with a great detail, and the results are evaluated with both software engineers of Siemens and academicians. One of the measures defined in our study, the number of vague phrases, was designed by considering the results of Gleich, et al. [139].

Fabbrini, et al. [144] presented a tool (QuARS) for automatic identification of low-quality requirements. The tool is based on a number of quality properties

and measurements subsumed into four quality categories: testability, completeness, understandability, consistency. The tool evaluates requirements based on a set of measures (indicators) of these properties. The five measures that they present – optionality, subjectivity, vagueness, weakness, implicity – are similar to our NV measure. Generally these are phrases that introduce vagueness or imprecision in requirements. From the study it is not clear how exactly these five measures are calculated, because they do not present an exhaustive list of keywords, but we observed that most of the keywords they present we have included in the measurement method of NV. Few keywords that they consider we have deliberately excluded from our list, such as, "fast" or "below", because they often do not cause any problem. They also define a measure called multiplicity which indicates the number of multiple statements in a requirement. To calculate this they count the number of "and", "or", and, as they say, similar phrases. We have generalized this measure by our NC measure which presents an exhaustive list of conjunctions to calculate the number of actions in a requirement.

In a very recent study Femmer, et al. [57] conducted a meticulous evaluation of eight categories of requirements' smells which are fundamentally based on eight types of ambiguity expressions. The evaluation in four real cases showed that many of the smells are often problems in practice, and moreover, they are automatically detectable. An interesting finding of them is that pure morphological analysis often introduces false positive results, and this finding is congruent with results of our paper. Femmer, et al. [151] also developed a tool which analyzes requirements based on eight types of ambiguous terms. The tool was applied in Daimler AG and showed that it can help practitioners with indicating problematic requirements.

Génova, et al. [137] proposed a framework and tool support (Requirements Quality Analyser, RQA) for quality improvements of textual requirements. They measure such properties as size, readability, punctuation, imprecise terms, verbal tense, number of versions, degree of nesting, overlapping, and dependencies of requirements. Then they develop a quality indicator for requirements based on these measures. The aforementioned two studies do not specify how exactly the measures are calculated and evaluated. They qualitatively evaluate the tools based on the feedback of companies that use it. As our results showed, several of the measures that they have used, such as number of versions, nesting degree, and punctuations are poor indicators and introduce noise in the measurement accuracy. Additionally it is important to select the independent measures among all designed measures, so several strongly correlated measures are not used in quality assessment formula at the same time. Considering the upper mentioned concerns we would recommend to be extra careful when designing measures and using them in a formula as a means of inference.

Parra, et al. [152] used measures presented in [137] to evaluate the correctness of requirements. They train a classifier based on the measures and experts' manual classification to identify "correct" and "incorrect" requirements. They achieve significantly high accuracy of classification compared with results of

manual assessors (83-87%). It is not clear from the paper whether the manual assessors were classifying correctness based on a predefined template or based on semantic understanding of requirements correctness. Nevertheless, the paper shows that it is possible to achieve high accuracy of automated quality checking, which is valuable for our research.

The previous studies have mainly focused on different types of ambiguity measures. They also have conducted evaluation to understand to what extent these measures can detect the problem areas. Our work is a continuation of the aforementioned studies. Our contribution to the state of art is the new measures of complexity and coupling, which can be used in combination with the previously created measures of ambiguity in order to enhance the precision of the problem detection. Moreover, we also provide an evaluation of the measures (and the method) based on not only how well the method indicates the problem areas but also how well the method can rank requirements based on the severity of the problems as ranked by the manual assessors.

12 Summary

This paper presents a method for proactive requirements reviews. The method is based on four internal quality measures of textual requirements, which were developed in an action research project conducted in a large software development organization. The evaluation results in four software development companies showed that the method can identify 73% - 80% of the needed improvements that software engineers working manually would identify. The method was deployed and used in the companies in different ways, considering their ways of working. The results also showed that more evaluation of the method is needed in order to understand the possibilities of its generalization across different sizes and domains of products.

PAPER 4

Evaluating Code Complexity Triggers, Use of Complexity Measures, and the Influence of Complexity on Maintenance Time

ABSTRACT

Code complexity has been studied intensively over the past decades because it is a quintessential characterizer of a code's internal quality. Previously, much emphasis has been put on creating code complexity measures and their application in practical contexts. To date, most measures are created based on theoretical frameworks, which determine the expected properties that a code complexity measure should fulfill. Fulfilling the necessary properties, however, does not guarantee that the measure characterizes the code complexity that is perceived by software engineers. Subsequently, code complexity measures often turn out to provide rather superficial insights into code complexity. This paper supports the discipline of code complexity measurement by providing insights into the code characteristics that trigger complexity, the use of code complexity measures in industry, and the influence of code complexity on maintenance time from an empirical perspective. Results of an online survey, conducted in seven companies and two universities with a total of 100 respondents are presented, and show that among other code characteristics, two such characteristics significantly increase code complexity, which subsequently have a major influence on the maintenance time of code. Notably, existing code complexity measures are poorly used in industry.

1 INTRODUCTION

The internal quality of software influences the ability of software engineers to progress software development. A major aspect of internal quality is the code complexity, which directly affects the maintainability and defect proneness of code [153, 154]. Therefore, research interest on the topic of complexity has been high over the years. Complexity measures have been designed to apply the notion of complexity practically [9, 22, 155]. Complexity measurement allows complexity to be quantified and its influence on aspects of code, such as maintainability and defect proneness, to be understood. The concept of complexity, however, is not an atomic concept, so it is difficult to design a single measure that quantifies complexity thoroughly. Instead, several complementary measures are designed to measure different aspects of complexity. Consequently, the insight that is provided by this combination of measures is expected to provide a fair assessment of the magnitude of complexity for a given piece of code.

Designing complexity measures often has followed theoretically established frameworks, according to which a complexity measure should either fulfill a predetermined set of properties or comply with a set of rules [27, 156-158]. Theoretical frameworks for creating measures are justifiably necessary because they propose a common foundation upon which complexity measures should be designed. Nevertheless, we believe that theoretical frameworks alone are unsatisfactory since detailed knowledge supporting the design of measures needs to be extracted from empirical data. Specifically, to design a complexity measure one must:

1. Identify specific code characteristics that should be considered for complexity measurement
2. Understand whether these characteristics are actually measurable in practice
3. Evaluate the contribution of these characteristics to complexity increase
4. Observe existing complexity measures and determine how well they capture code characteristics that influence complexity
5. Evaluate the usefulness and popularity of existing complexity measures in practice
6. Assess the influence of complexity on code maintainability.

Since these factors are not addressed fully in the design of complexity measures, existing measures are usually perceived as being only moderately accurate in complexity measurement. A typical example of this is when two source code functions have the same cyclomatic complexity value. The cyclomatic complexity is the same, but intuitively we understand that one of the functions is more complex because, for example, it has more nested blocks [159]. These kinds of issues are apparent in many well-recognized complexity measures and have been discussed previously [72, 160-162]. In practice, certain modules of code

are perceived to be intrinsically more complex and, therefore, more difficult to maintain despite their relatively small size [30, 163].

We believe that the aforementioned knowledge required for designing measures can be partially or fully answered if we consider the collective viewpoint of software engineers, which would provide an insightful contribution for academics when designing complexity measures and measurement-based prediction models.

The aim of this study, therefore, was to acquire such knowledge using the following five research questions (RQ):

RQ 1: Which code characteristics are perceived by software engineers to be the main triggers of complexity?

RQ 2: How frequently are complexity measures used in practice?

RQ 3: How strongly do software engineers perceive complexity to influence code quality?

RQ 4: How much does complexity affect maintenance time?

RQ 5: Do the responses to RQ 1 to RQ 4 depend on the demographics of respondents?

Here, we present the evaluation results of code characteristics as complexity triggers, the extent to which complexity measures are used in industry, and the evaluation results of complexity influence on internal quality and maintainability based on a survey of 100 software engineers. The survey included both structured and open questions that allowed for comments. In summary, the results of the five categories of questions (RQ1–RQ5) showed that:

1. Of the eleven proposed code characteristics, only two markedly influence complexity growth: the nesting depth and the lack of structure
2. None of the suggested nine popular complexity measures are actively used in practice. Size and change measures as forms of complexity measures are used relatively more often, although not for complexity or quality assessment purposes
3. Complexity is perceived to have strong negative influence on aspects of internal quality, such as readability, understandability and modifiability of code
4. The statistical mode (most likely value) of the software engineers' assessment indicates that complex code requires 250–500% more maintenance time than simple code of the same size
5. The demographics of the respondents did not influence the results of RQ 1–RQ 4.

These results suggest that managing complexity has a crucial role in increasing product quality and decreasing maintenance time. Moreover, the results provide insight as to which code characteristics should be considered in code complexity

measurement and management. Importantly, however, the reasons why complexity measures are not actively used in complexity management activities need thorough investigation.

2 THE LANDSCAPE OF CODE COMPLEXITY SOURCES

The term *complexity* has been used widely in many disciplines, usually to describe an intrinsic quality of systems that strongly influences human understandability of these systems. Unfortunately, as no generally accepted definition of complexity that would facilitate its measurement exists, every discipline has its own rough understanding on how to quantify complexity.

C*ode complexity*, the subject of this study, is not an exhaustively defined concept either. In the IEEE standard computer dictionary, code complexity is defined as "the degree to which a system or component has a design or implementation that is difficult to understand and verify" [21]. According to Zuse [9], the true meaning of code complexity is the difficulty in understanding, changing and maintaining code. Fenton and Bieman [22] view code complexity as the resources spent on developing (or maintaining) a solution for a given task. Similarly, Basili [8] defines code complexity as a measure of the resources allocated by a system or human while interacting with a piece of software to perform a given task. An understanding of how to measure complexity and make code less complex is not facilitated by these definitions because they focus on the effects of complexity, i.e., the time and/or resources spent or experienced difficulty, and thus do not capture essence of complexity. Briand, et al. [23] have suggested that complexity should be defined as an intrinsic attribute of code and not its perceived difficulty by an external observer, which would indeed aid the understanding of the origin of complexity.

To outline a landscape of the source of code complexity that would facilitate the design of the survey questions and the interpretation of the results, we adopted a general definition of system complexity that considers it to be an intrinsic attribute of a system. An example of such a definition is provided by Moses [25], who defines complexity as "an emergent property of a system due to its many elements and interconnections". This is very similar to the definition of Rechtin and Maier [6], stating that "a complex system has a set of different elements so connected or related as to perform a unique function not performable by the elements alone". These two definitions are suitable for understanding and measuring code complexity because they indicate the origin of complexity, namely different elements and their interconnections in the code. Elements and interconnections appear to be the direct sources of code complexity, i.e., those sources that directly influence code complexity and thus complexity measurement. Based on these two definitions, we can imply three things: (i) The more elements and interconnections the code contains, the more complex the code; (ii) Since the elements and interconnections always have some kind of representation (for reading, understanding and interpreting), the complexity depends on

this representational clarity; and (iii) If we consider that any system usually evolves over time, the evolution of elements and interconnections also determines a change in complexity.

Considering these three points, we postulate that there are three direct sources of code complexity:

1. Elements and their connections in a unit of code
2. Representational clarity of the elements and interconnections in a unit of code
3. Evolution of a unit of code over development time.

Elements and their connections: Complexity emerges from existing elements and their interconnections in a unit of code. For a unit of code, the elements are different types of source code statements (e.g., constants, global and local variables, function calls, etc.). The interconnections of elements can be expressed both by mathematical operators (e.g., addition, division, multiplication, etc.) and control statements, Boolean operators, pointers, nesting level of code, etc. Each type of element and each type of connection increases the magnitude of code complexity to a different extent.

Representational clarity: Complexity arises from unclear representation of the code. This is concerned with how clearly the elements and interconnections are presented to demonstrate their intended function. This means that there could be a difference between what a given element does and what its representation implies that it does. A typical example is using misleading names for functions and variables.

Intensity of evolution: Code evolution can be characterized by the frequency and magnitude of changes of that code. Evolution of the code is also regarded as a source of complexity because this changes the information about how a given piece of code operates in order to complete a given task. If a software engineer already has knowledge on how the code operates, then the evolution of the code will partly or completely destroy that knowledge because changes will introduce a new set of elements and interconnections into the code. *This does not imply that changing the code always makes the code more complex, it only implies that the level of complexity, solely driven by changes in the code, increases.* At the same time, the level of complexity that emerges from elements and their connections might decrease and thus potentially reducing overall complexity. This occurs often in practice when the code is refactored successfully.

We used these three direct sources of complexity to correctly identify those code characteristics that belong to any of these sources as direct complexity triggers. Subsequently, we developed the survey questions to evaluate these characteristics.

These three sources of complexity comply with the definitions of Moses [25] and Rechtin and Maier [6], and are directly observable on the code, hence our term "direct sources of complexity". In addition, there are several other, indirect sources of complexity, such as those described by Mens *et al*. [5]. These are not

directly visible in the source code, although they somewhat influence complexity. Examples include the:

1. Complexity of the problem to be solved by the program
2. Selected design solution for the given problem
3. Selected architectural solution for the given problem
4. Complexity of the organization where the code is developed
5. Programming language
6. Knowledge of developers in programming
7. Quality of the communication between developers and development teams
8. Managerial decisions
9. Domain of development.

In this study, we did not consider the indirect sources of complexity, their measures and influence on the internal quality; this requires additional study.

In summary, we perceive complexity to be an emergent property of code that is magnified by the addition of more elements and/or interconnections, changing the existing ones or not clearly specifying the function of existing elements. We consider that the origin of code complexity is outlined primarily by the three aforementioned sources. Since the other factors are not direct sources of complexity, they should not be included in the landscape of code complexity sources.

3 Research Design

To address the research questions (RQ 1–RQ 5), we conducted an online survey [49] with software engineers in Industry and Academia. Most data were collected using structured questions of which there were 25 in total organized as a six-point Likert scale. An even number for the scale values avoided a scale midpoint, thereby ensuring that respondents could choose a higher or lower estimate than average. Additionally, three open questions allowed respondents to add choices that might otherwise have been missed in the structured questions. The survey consisted of five logical parts:

1. Part 1: Identified the demographics of participants
2. Part 2: Estimated the extent to which different code characteristics make code complex
3. Part 3: Evaluated the influence of complexity on internal code quality attributes
4. Part 4: Evaluated the most commonly used complexity measures in industry
5. Part 5: Assessed the influence of complexity on development time.

Survey participants were software engineers from seven large software development organizations and two universities, all of which were a part of a research consortium called Software Centre. The seven collaborating companies were: Ericsson, Volvo Group, Volvo Car Group, Saab and Axis Communication all

from Sweden, Grundfos from Denmark, and the Swedish branch of Jeppesen in the United States. The two universities were University of Gothenburg and Chalmers University of Technology The companies represented a variety of market sectors, namely telecommunication equipment, trucks and cars, the air defense system, video surveillance system, pumps, and the air traffic management system. All of these systems used both small and large software products, which had been developed using different development processes, such as Lean, Agile, and V-model. Complexity was an actively discussed topic in these companies so many software engineers were motivated to participate in the survey. Since we were involved in previous research with these companies and knew the products and development organizations, we found this selection of companies rational from the perspective of construct validity.

We shared the online address of the survey with the collaborating managers or organizational leaders in the companies, who then distributed the survey within their corresponding software development organizations, targeting software engineers who worked intensively with software development. Our objective was to collect at least 100 responses from the companies in order that one answer should represent at most one percent. One initial request and one reminder were sent to prompt a response from the participants. In total, however, 89 responses were received from the companies. In addition, 11 responses were received from the two universities. We selected university respondents who worked in close collaboration with software companies and had developed software products themselves earlier in their careers. In contrast to the companies, the survey link was distributed in universities directly to potential respondents. The response rate was estimated by counting the number of potential respondents who received the survey link from corporate contacts and directly from us. Approximately 280 people received the survey link, 100 of whom responded, resulting in a response rate of approximately 36%.

To minimize any misunderstanding of words or concepts in the survey questions, two pilot studies were conducted prior to the survey launch. Feedback from a group of nine software engineers from companies was also used to improve the survey and the choice of assessment scales. This test group was also asked to interpret their understanding of the survey questions in order to identify any misinterpretations. The survey was only launched once all nine engineers understood the survey questions as they were intended to be understood. The results of the pilot studies are not included in the results of this study.

3.1 Demographics and the Related Questions

The first part of the survey investigated the participant demographics. Five fields were given for information related to demographics, as presented with the specified options in Table 1. Data for the five fields were collected using the following five statements:

1. Select your education

2. Select your job title
3. Select your domain
4. Select the years of experience that you have in software development
5. Select the programming languages that you usually work with.

Table 1 Specified fields and options for acquiring demographic data

Education	Job Title	Domain	Experience	Programming Language
Computer Science (31)	Developer (49)	Telecom (30)	< 3 years (10)	Python / Ruby (30)
Software Engineering (37)	Tester (5)	Automotive (23)	3–5 years (11)	Java / C# (43)
Information systems, Informatics (7)	Architect (13)	Defence (10)	6–10 years (20)	C++ (42)
Computer Engineering (11)	Team leader, Scrum master (14)	Enterprise Systems (14)	11–15 years (20)	C (57)
Management (2)	Product owner (2)	Web Development (2)	> 15 years (40)	JavaScript/PHP (15)
Economics (0)	Project manager (1)	Health Care (0)		Perl / Haskell (10)
Electrical, Electronic Engineering (38)	Researcher (12)	Academia (11)		TTCN / Tcl / Shell (11)
Other (12)	Other (4)	Other (16)		Other (21)

In the cases of "Job Title" and "Experience", options were given by radio buttons with a "one-choice-only" option. Checkboxes were specified for all other fields to enable respondents to select more than one option. In Table 1, the number of responses obtained per demographical category is shown in brackets. In addition, graphical representations of these results can be found in Section 4.1.

3.2 Selected Code Characteristics as Complexity Triggers

The second part of the survey concerned code characteristics with the objective of understanding the extent to which each code characteristic increases code complexity. We proposed eleven code characteristics that can potentially increase code complexity based on our previous study conducted with Ericsson and Volvo Group [164]. In that study, we were designing code complexity measurement systems for these companies in which approximately 20 software engineers were involved. Based on regular discussions on topics like the origin of complexity and which code characteristics are usually considered in complexity measurement, we determined the eleven common code characteristics that

were used in this study. These characteristics belong to one of the three main sources of the complexity landscape presented in Section 2. The three main sources, complexity characteristics and their descriptions are shown in Table 2.

Table 2 Code characteristics and descriptions

Three sources of complexity	11 Code Characteristics	Description of the Characteristic
Elements and interconnections	Many operators	All mathematical operators (e.g., =, +, -, /, mod, sqrt)
	Many variables	Both local and global variables in the code
	Many control statements	Control statements in the code (e.g., “if”, “while”, “for”, etc.)
	Many calls	All unique invocations of methods or functions in the code
	Big nesting depth	The code is nested if there are many code-blocks inside one another
	Multiple tasks	Logically independent tasks that are solved in one code unit
	Complex requirement specification	This relates to detail requirement specification that the developers use to design software
Representational clarity	Lack of structure	This relates to correct indentations, proper naming and using the same style of coding for similar patterns of code
	Improper or not existing comments	This relates to code that does not have any comments or the existing ones are misleading
Evolution	Frequent changes	This relates to code that changes frequently thus behaving differently over development time
	Many developers	This relates to code that is modified by many developers in parallel

Nine of the code characteristics are easily observable in the code. Although two of the characteristics—“complex requirement specification” and “many developers”—are difficult to observe in the code, they still directly influence complexity:

1. Many requirements in industry are written in a very detailed manner, such as pseudocode or detail diagram. Such detail specifications do not allow developers to consider the design of the code, but merely translate the specification into a programming language so the specification complexity is largely transferred into the code.
2. Many developers who make changes on the same piece of code add a new dimension on the code change as a type of complexity. The information needed to learn about the change in this case comes from multiple developers.

To investigate the effect of these eleven characteristics on code complexity, one statement (question) per characteristic was formulated be answered using the specified Likert scale. For example, the statement for function calls is shown Figure 1. The three dots at the end of the statement were to be completed by one of the options given in the Likert scale. The second line explained in more detail what was meant by the given characteristic to ensure no uncertainty on the part of the respondent.

Generally many calls in a unit of code make that code ...

We consider all unique invocations of methods or functions in the code

- not complex at all
- little complex
- somewhat complex
- rather complex
- quite complex
- very complex
- N/A

Figure 1 Example of a question regarding a given code characteristic

The rest of the statements about code characteristics were organized the same way as that shown Figure 1. In most of the statements, we intentionally emphasized that "*many* of something" makes code complex, i.e., *many* operators, *many* variables, *many* control statements, etc. In other statements, we used different methods of framing, for example, the lack of structure, the frequent changes, etc. At the end of this part of the survey, an open question was included to allow respondents to suggest other code characteristics that they believed could significantly increase complexity.

3.3 Complexity and Internal Code Quality Attributes

The third part of the survey was designed to acquire information on the extent to which code complexity influences internal code quality attributes that are directly experienced by software engineers. Note that by internal code quality attributes we do not mean the emergent properties of software code, such as size, length, cohesion, coupling and complexity itself, but the quality attributes that arise from the relationship between the intrinsic properties of code and cognitive capabilities of engineers, namely readability, understandability and modifiability. We added "ease of integration" to these three attributes, however, since we consider it also plays an important role in code development and can be influenced by complexity. The four internal code quality attributes and their brief descriptions are shown in Table 3. The questions were organized to have six possible values of the Likert Scale plus an additional option to allow a "no answer" option. One of the four questions is shown in Figure 2 and depicts the organization of the rest of the questions.

Table 3 Internal quality attributes and descriptions

Internal code quality attributes	Explanation
Readability	The visual clarity of code that determines the ease for reading the code
Understandability	The conceptual clarity and soundness of code that ease the process of understanding the code
Modifiability	The logical soundness and independence of code that determine the ease of modifying the code
Ease of integration	The ease of merging a piece of code to a code development branch or to the whole product

In this section, only selected internal code quality attributes concerned with cognitive capabilities of the engineers working with the code were covered. Other internal code quality attributes, such as error-proneness or testability were not considered in this study because they are not directly experienced by software engineers when working with the code.

How much does the complexity influence the readability of a code unit?

- not at all
- a little
- somewhat
- rather
- quite
- very much
- N/A

Figure 2 Example of a question regarding the influence of complexity on internal quality

3.4 Selected Complexity Measures

The fourth survey section investigated the use of the most actively investigated complexity measures from the literature. Measures were selected based on their popularity in the literature, and particularly how often they are used for maintainability assessment and defect predictions.

The measures and their descriptions are shown in Table 4. To acquire information on the use of the measures, the frequency of use was assessed using a Likert Scale; an example of these questions is shown in Figure 3. The last option in this this “multiple choice” question was “never heard of it”, which essentially differs from that of “never used it” because in the former case, the reason why the measure is not used differs substantially from the latter. If a respondent selects “never heard of it”, this implies that the measure could be either useful for them or not; however, if the respondent has never heard of it, no conclusion can be made. In contrast, if a respondent answers “never used it” (or “hardly

ever used it") this might indicate a problem with the measure itself. An additional field was included at the end of this section that allowed respondents to add more complexity measures, which they did, but was not included in our list.

Table 4 Selected measures and descriptions

Name of the Measure	Description
McCabe's cyclomatic complexity [106]	The number of linearly independent paths in the control flow graph of code. This can be calculated by counting the number of control statements in the code
Halstead measures [165]	Seven measures completely based on the number of operators and operands
Fan-out [166]	The number of unique invocations found in a given function
Fan-in [166]	The number of calls of a given function elsewhere in the code
Coupling measures of Henry and Kafura [166]	Based on size, fan-in, and fan-out
Chidamber and Kemerer OO measures [167]	Inheritance level and several size measures for class
Size measures	Lines of code, number of statements, etc.
Change measures, e. g., Antinyan, et al. [140]	Number of revisions, number of developers, etc.
Readability measures, e. g., Tenny [168], Buse and Weimer [169]	Line length, indentations, length of identifiers, etc.

I use fan-out (the number of function calls in a given unit of code)

- Daily
- Weekly
- Monthly
- Hardly ever
- Never used it
- Never heard of it

Figure 3 Example of a question regarding the use of measures

3.5 Complexity and Maintenance Time

The fifth part of the survey concerned the influence of the code complexity on code maintenance time. Here the objective was to obtain quantitative information on how much time is spent unnecessarily on maintaining complex code. The quantitative information was based purely on a perceptive estimation of respondents; therefore, we expected the summary of the answers to be a rough estimation. The only question posed in this section is shown in Figure 4.

Generally a complex piece of code takes me X% more time of maintenance compared with a simple piece of code of the same size

Choose a rough value for X% from the provided list by your perception and experience.

- < 10 %
- 10 - 25 %
- 25 - 50 %
- 50 - 100 %
- 100 - 150 %
- 150 - 250%
- 250 - 500 % (2.5-5 times more)
- 500 - 1000 % (5-10 times more)
- Other:

Figure 4 Question investigating the influence of complexity on maintenance time

The question assumes that two pieces of code of the same size can differ significantly in complexity. The respondents were expected to estimate the additional time required to maintain a piece of complex code compared to the maintenance time of simple code of the same size. The answer was not expected to be based on any quantitative estimation, but rather on the knowledge and experience of respondents. At the end of this question, a field for free comments on respondents' thought processes when making the estimates was added.

3.6 Data Analysis Methods

Data was analyzed using descriptive statistics and visualizations. As regards descriptive statistics, percentages and statistical modes were used, whilst visualizations included tables and bar charts to summarize data related to the code characteristic, the use of complexity measures and the effect of complexity on the internal quality of code. Color-coded bars were used to enhance graph readability. Pie charts were used to visualize the complexity influence on maintenance time. The fields that had been specified for free text were analyzed by classifying answers into similar categories. As regards the code characteristics, the number of respondents who proposed a specific characteristic to be a significant complexity trigger was counted. With respect to measures, the number of respondents who mentioned a specific complexity measure that they used was not included in our list. The assessment of complexity influence on maintenance time was done by listing the tactics used by respondents for their assessment, as well as counting the number of respondents per proposed tactic.

In addition to the aforementioned analysis, across-sectional data analysis was also conducted to investigate whether the demographics of the respondents significantly influenced the results. We hypothesized that demographics do not influence the results and conducted statistical tests to either reject or confirm this hypothesis. Since the number of responses was only 100, it was not possible

to divide the data into many groups to obtain meaningful results because some groups had too few data for clear statistical analysis. Data, therefore, were divided into fewer groups for such analyses.

Table 5 shows all the possibilities for cross-sectional data analysis. The first row depicts the four main categories of data. The first column shows the five units of demographics. Every cell of the table indicates whether cross-sectional analysis for a pair of "demographical data" and "category of result" was conducted in this study. Three of the survey questions in the Demographics Section were specified by checkboxes. These questions concerned *education, domain* and *programming language.* Since these were specified by checkboxes, one respondent could select several choices concurrently, such that a statistical test to analyse the effect of demographics on the results could be conducted. The results concerning *complexity measures* and *complexity influence on code quality attributes* were so polarized over the categorical values that it was not possible to do any cross-sectional data analysis for these two categories either.

Table 5 Cross-sectional data analysis table

		Four categories of results			
Demographics		Complexity characteristics	Complexity influence on quality attributes	The use of complexity measures	The influence of complexity on maintenance time
	Education	✗	✗	✗	✗
	Job	✓	✗	✗	✓
	Domain	✗	✗	✗	✗
	Experience	✓	✗	✗	✓
	Prog. language	✗	✗	✗	✗

The remaining four cells of Table 5, however, show the four pieces of cross-sectional analysis that were done. Methods for each of the analyses are presented in the following subsections.

3.6.1 Evaluating the Association between Job Type and Assessment of Code Characteristics

Here, the objective was to understand whether the type of job has any association with the assessment results of code characteristics. Therefore, the type of job was divided into two groups, developers and non-developers. Developers were respondents who marked their role as "developer" in the survey, whilst the non-developers were those respondents who marked any role other than "developer". Hence, the variable *type of job* has two possible categorical values: developer and non-developer.

Table 6 Original six values and derived two values of "assessment" for code characteristic

Original values	Derived values
Not complex at all	Little influence
Little complex	
Somewhat complex	
Rather complex	Much influence
Quite complex	
Very complex	

This division of jobs is motivated by the fact that developers work directly with the code and they themselves influence code complexity, whereas non-developers are only influenced by the code complexity. Therefore, we expected a statistical difference between these two groups. Similarly, we derived two values of "assessment" for code characteristic from the original six values. The original six values and the derived two values are presented in Table 6. The two values for both "type of job" and "assessment" allowed us to develop a contingency table based on which Chi-Square test was conducted to determine whether there was a statistical difference in assessment of code characteristics by people with different jobs. An example is given for *nesting depth* in Table 7, which shows that nine developers indicated that *nested* code has little influence on complexity increase, while 42 developers indicated the opposite.

Table 7 Contingency table for "type of job" and "assessment of code" characteristics

	Developer	Non-developer
Little influence	9	7
Much influence	42	41

Because the variables can have categorical values, the Chi-Square test was used to assess whether the type of job and assessed influence were associated. To perform this analysis for all eleven code characteristics, eleven tables similar to that of Table 7 were developed

3.6.2 Evaluating the Association between Experience and Assessment of Code Characteristics

Two values of "experience" from the original five values were derived to conduct this analysis (Table 8), namely, "much experience" and "little experience". The analysis was conducted in exactly the same way as in Section 3.6.1.

Table 8 The original five-scale assessment and the derived two-scale assessment

Original values	Derived values
< 3 years	Little experience
3–5 years	
6–10 years	Much experience
11–15 years	
> 15 years	

3.6.3 Evaluating the Association between Type of Job and Assessment of Complexity Influence on Maintenance Time

Two values for "type of job" (developer and non-developer) were used to analyse the effect of complexity on maintenance time. Originally, the variable that showed the assessment values of complexity influence on maintenance time had eight categorical values, but to ensure sufficient data points for a meaningful Chi-Square test, three categorical values were derived from these eight values. Three values were chosen because eight might be too many to categorize them into two values as with the other variables. The original eight values and the three derived values are shown in Table 9.

Table 9 Original eight values of assessment and three derived values of assessment

Original values	Derived values
0–10%	Little influence
10–25%	
25–50%	
50–100%	Much influence
100–150%	
150–250%	Very much influence
250–500%	
500–1000%	

Based on the three values of "assessment" in Table 9 and two values of "type of job", a contingency table was made (Table 10). Using this table, a Chi-Square test was performed to determine whether there was a significant difference between "assessment" and "type of job".

Table 10 Cross-sectional data for "type of job" and assessment of "complexity influence on maintenance time"

	Developer	Non-developer
Little influence	8	10
Much influence	13	16
Very much influence	29	22

3.6.4 Evaluating the Association between Experience and Assessment of Complexity Influence on Maintenance Time

Based on two values of “experience” and three values of “complexity influence on maintenance time”, a contingency table was developed (Table 11). A Chi-Square test using data in Table 11 was done to determine whether there was an association between “experience” and assessed “complexity influence on maintenance time”.

Table 11 Cross-sectional data for “experience” and assessment of “complexity influence on maintenance time”

	Little experience	Much experience
Little influence	3	15
Much influence	7	22
Very much influence	11	40

4 RESULTS AND INTERPRETATIONS

The results are divided into six sections. The first section shows demographic data of all respondents. The subsequent four sections present results on (i) code characteristics, (ii) complexity influence on internal quality, (iii) the use of complexity measures in industry, and (iv) the influence of code complexity on the maintenance time of code. These four sections answer the first four research questions (see RQ 1–RQ 4 in Introduction). Section 6 shows the cross-sectional data analysis when slicing data according to the demographic data and answers the fifth research question (RQ 5).

4.1 Summary of Demographics

This section presents data from the five demographical dimensions of the respondents, i.e., the type of *education* of respondents, the type of *job* the respondents had, *the* software development *domain* the respondents worked in, and the group of *programming languages* they used.

The educational background of the respondents is shown in Figure 5. Since this question was based on checkboxes, respondents could select multiple answers. In total, 100 respondents gave 138 ticks, indicating that several respondents had more than one educational background. Figure 5 shows that the majority of respondents had received education in electrical/electronic engineering, software engineering or computer science. The popularity of electrical/electronic engineering can be explained by the fact that many respondents were from car and pump industries, which traditionally demand competence in electrical engineering. The increasing importance of software in these industries has created a favorable environment for electrical engineers to become software development

specialists over time. Figure 6 shows the job titles of respondents. Almost half of respondents (n = 49) are developers, but there was also a number of architects, development team leaders and researchers.

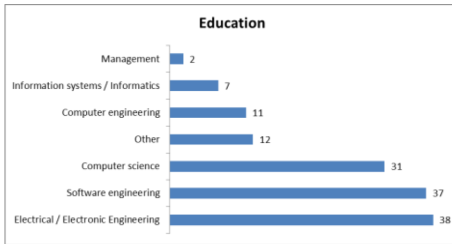


Figure 5 Respondents' educational background

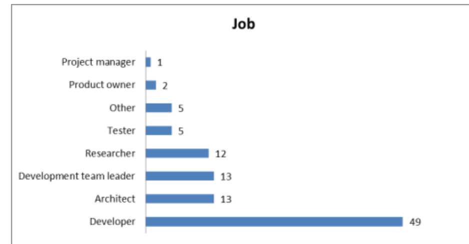


Figure 6 Respondents' job description

This was unexpected in that there were few "testers" among the respondents, although this could be explained by the fact that many respondents are working in Agile development teams, which have no specific testers and developers. Notably, these two jobs ("testers" and "developers") often are interchangeable and both are known as "developers" in some organizations.

Figure 7 presents the domain of respondents. In total, 105 answers were given by the 100 respondents, i.e., five or fewer respondents had worked in more than one domain. Fifty three respondents alone worked in the telecom and automotive domains.

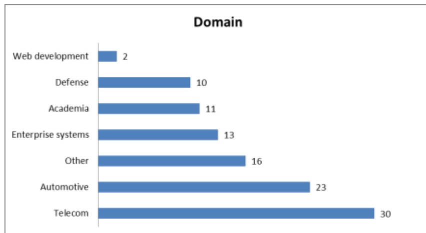


Figure 7 Software development domain of respondents

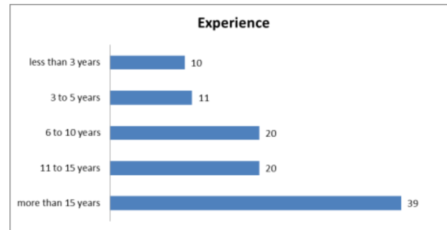


Figure 8 Experience of respondents in software development

Figure 8 presents the experience of respondents in software development. Thirty nine respondents had more than 15 years of experience and, generally, only 10 respondents had little experience (less than 3 years).

Finally, Figure 9 shows the programming languages used by respondents. According to the responses, C language was dominant in these industries, partly due to embedded development and partly because all products were old and mature having been developed for many years and traditionally relying on C language.

In total, 100 respondents gave 171 ticks for programming languages, indicating that many of the developers used several programming languages.

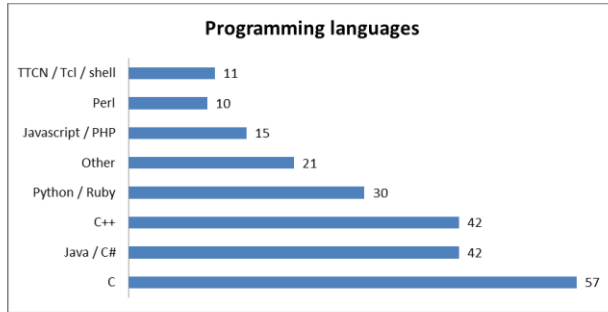


Figure 9 Programming languages used by respondents during their entire experience

4.2 Code Characteristics as Complexity Triggers

Figure 10 shows the eleven characteristics and their evaluated influence on complexity. The vertical axis shows the number of respondents, and every bar represents one characteristic. Bars are color-coded, with the darkest red indicating that the given characteristic made code very complex. The darkest green color indicates that the given characteristic does not make the code any complex.

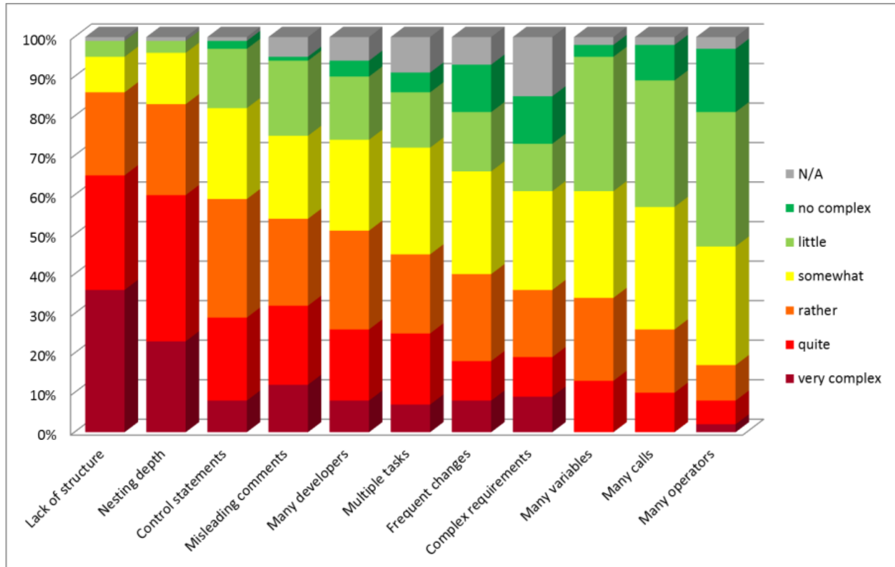


Figure 10 Influence of code characteristics on complexity

Overall, it can be inferred from Figure 10 that two characteristics—the *lack of structure* and *nesting depth*—are separated from all other characteristics due to their estimated magnitude of influence. If the red-orange area is considered to be an area of *major influence,* then the majority of respondents (over 80%) be-

lieved these characteristics to have *major influence* on complexity. The influence level of the rest of the characteristics decreased gradually along the horizontal axis. Approximately 50–60% of respondents believed that *control statements, misleading comments* and *many developers* have a major influence on complexity, whilst about 35–45% of respondents believed that *multiple tasks, frequent changes* and *complex requirements* did so.

The larger grey area for "Complex Requirements" may indicate that respondents found it difficult to evaluate this factor's influence on complexity. The exact numbers of estimates are presented in Table 12. The statistical modes of the evaluations per characteristic are emphasized by color so that what values the modes have on the assessment scale area easy to read. This alternative representation of the results enables greater understanding of the influence of characteristics on complexity. The characteristics have been divided into four groups based on their modes. The most influential characteristics are *nesting depth* and *lack of structure,* the modes of which reside in the categories of *very complex* and *quite complex.* The next three characteristics have modes categorized as *rather complex*, making them the second most influential characteristics. The rest of the characteristics are interpreted similarly.

Table 12 Influence of code characteristics on complexity with the modes emphasized

	very	quite	rather	some.	little	no
Lack of structure	36	29	21	9	4	0
Nesting depth	23	37	23	13	3	0
Control statements	8	21	30	23	15	2
Misleading comments	12	20	22	21	19	1
Many developers	8	18	25	23	16	4
Multiple tasks	7	18	20	27	14	5
Frequent changes	8	10	22	26	15	12
Complex requirements	9	10	17	25	12	12
Many variables	0	13	21	27	34	3
Many calls	0	10	16	31	32	9
Many operators	2	6	9	30	34	16

In addition to the evaluation of code characteristics, respondents were also able to provide qualitative feedback on what other characteristics they considered might significantly influence code complexity. Eight respondents mentioned that it is preferable to separate categories of *"missing comments"* and *"misleading comments"* since they influence complexity differently, i.e., missing comments are not considered a problem if the code is well-structured and written in a self-

explanatory manner; however, misleading comments can significantly increase the representational complexity of the code.

One respondent stated that it is always good practice to incorporate the comments into the names of functions, variables, etc. because it is highly likely that over time and with the evolution of software, comments become misleading because they are not always updated.

Four respondents mentioned that they prefer global and local variables to be separated since global variables introduce significantly higher complexity than local variables. According to respondents, the extensive use of global variables can cause high complexity and decrease the ability to find serious defects. A case study conducted in Toyota also supports this line of argument [170].

Three respondents mentioned that multiple levels of inheritance with functions overloaded at many different levels can significantly increase complexity. In such cases, it is hard to understand which piece of code is actually executed. Another three respondents mentioned that the extensive use of pre-processors, macro-code and many levels of pointers can also significantly influence complexity.

As well as comments regarding code characteristics, respondents also reflected on other issues of code complexity. For example, several recognized that there are two types of complexity: essential and accidental, the former being inherent to the problem and the latter arising from non-optimal methods of programming, and that sometimes it is difficult to understand whether the complexity is essential or accidental.

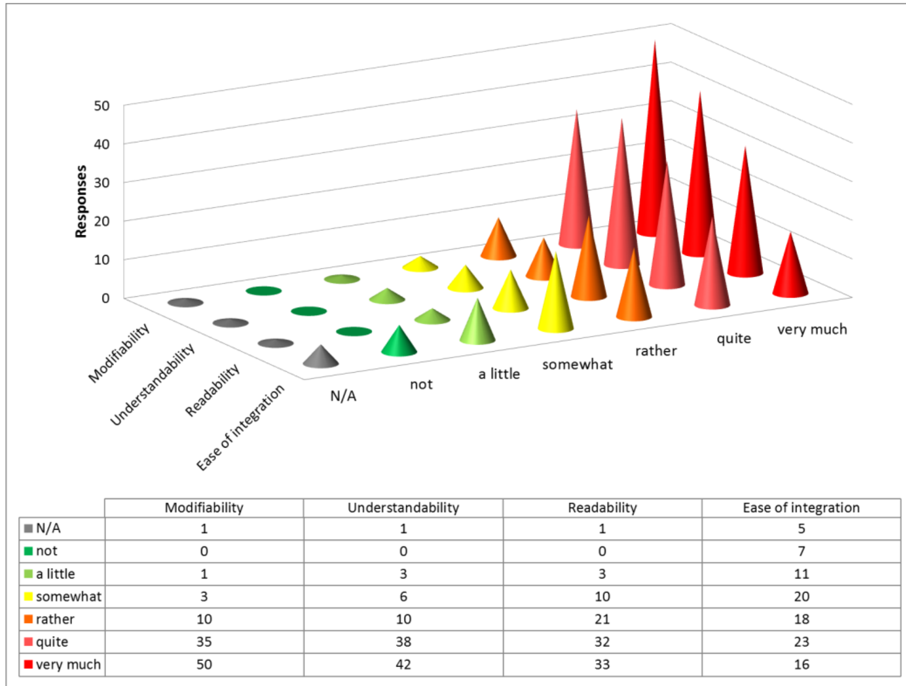
4.3 The Influence of Complexity on Internal Code Quality Attributes

This subsection presents the negative influence of code complexity on internal code quality attributes, such as *readability, understandability, modifiability* and *ease of integration.* Figure 11 shows the evaluation results for the influence of code complexity on internal code quality attributes. The diagram shows that the majority of respondents agree that complexity has a huge influence on three attributes: *readability, understandability* and *modifiability.*

Modifiability, which can be considered the essential constituent of code maintainability, is influenced by complexity the most. Ninety five respondents believed that the complexity has major influence on code modifiability. Only four respondents believed otherwise, and one respondent did not answer the question. Every cell of the table in Figure 11 shows the number of responses obtained per pair of internal code quality attribute and magnitude of influence, and the first row shows the *"N/A"* option.

The last three rows of this table tend to show greater numbers than the first three rows, indicating that the huge influence of complexity on internal code

quality attributes. One of the attributes, *"ease of integration"*, is believed not to be influenced by complexity as much as the other three, which is intuitive because integration often concerns making the specified piece of code work with the rest of code without understanding its content in detail, and thus without actually dealing with complexity.



	Modifiability	Understandability	Readability	Ease of integration
N/A	1	1	1	5
not	0	0	0	7
a little	1	3	3	11
somewhat	3	6	10	20
rather	10	10	21	18
quite	35	38	32	23
very much	50	42	33	16

Figure 11 Influence of code complexity on internal code quality attributes

4.4 The Use of Complexity Measures

The use of code complexity measures in industry is presented here. Nine complexity measures (or groups of complexity measures) and their popularity are presented in Figure 12. Figure 12 also shows that the next three measures (McCabe's cyclomatic complexity, fan-in, and fan-out) were slightly used. Only two groups of measures (*size measures* and *change measures*) were moderately used by respondents, although this does not necessarily mean that they were used as a means of quality assessment, but for other purposes, such as effort estimation or productivity measurement. Table 13 presents a more detailed view of the use of these measures. The modes of the first five measures in the table indicate that many respondents had never heard of the specified measures. The rest of the measures appear to be known by many, but never used in any systematic way.

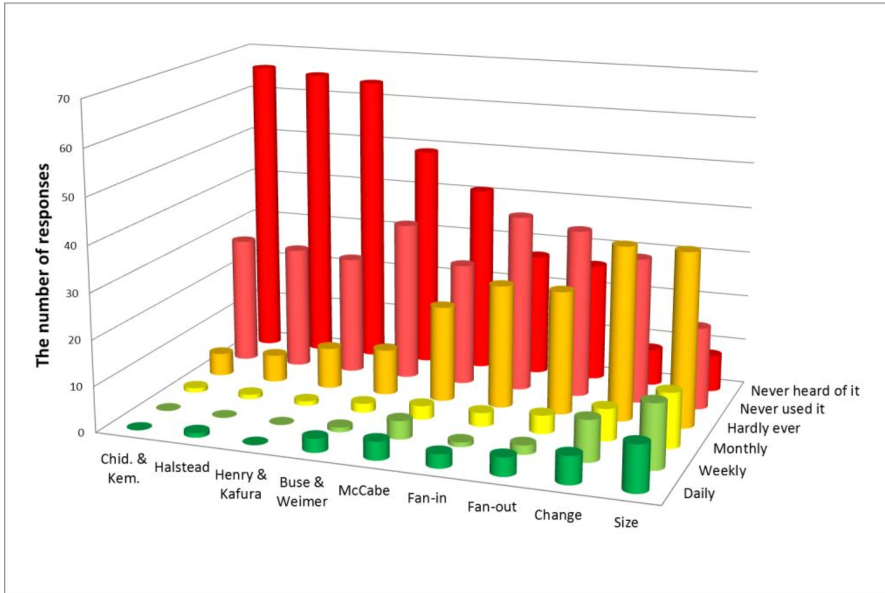


Figure 12 Use of complexity measures in industry

Figure 12 also shows that the next three measures (McCabe's cyclomatic complexity, fan-in, and fan-out) were slightly used. Only two groups of measures (*size measures* and *change measures*) were moderately used by respondents, although this does not necessarily mean that they were used as a means of quality assessment, but for other purposes, such as effort estimation or productivity measurement.

Table 13 presents a more detailed view of the use of these measures. The modes of the first five measures in the table indicate that many respondents had never heard of the specified measures. The rest of the measures appear to be known by many, but never used in any systematic way. Figure 12 also shows that the next three measures (McCabe's cyclomatic complexity, fan-in, and fan-out) were slightly used. Only two groups of measures (*size measures* and *change measures*) were moderately used by respondents, although this does not necessarily mean that they were used as a means of quality assessment, but for other purposes, such as effort estimation or productivity measurement. Table 13 presents a more detailed view of the use of these measures. The modes of the first five measures in the table indicate that many respondents had never heard of the specified measures. The rest of the measures appear to be known by many, but never used in any systematic way.

Table 14 more concisely represents the data, classifying the frequency of use into three categories: *regularly used, not used,* and *never heard of it.* We consider a measure is used regularly if it is used *daily*, *weekly*, or *monthly*, and a measure is *not used* if it is classified as *hardly ever* or *never used it.*

Table 13 Measures and their use represented by statistical modes

	Daily	Weekly	Monthly	Hardly ever	Never used	Never heard of
Chidamber & Kemerer	0	0	1	5	28	66
Halstead	1	0	1	6	27	65
Henry & Kafura	0	0	1	9	26	64
Buse & Weimer	3	1	2	10	35	49
McCabe	4	4	3	21	27	41
Fan-in	3	1	3	27	39	27
Fan-out	4	2	4	27	37	26
Change	6	9	7	38	32	8
Size	10	14	12	38	18	8

The latter two categories mean that respondents knew of the measure and had even have tried to use it, but for some reason did not consider using it regularly. We have ascertained reasons for this through informal talks from software engineers in the participating companies; these vary and are inconclusive. For example:

1. Company regulations either do not consider using the measure or another measure is the accepted standard
2. Developers do not believe that use of the measure can compensate for the time spent on the measurement
3. The measure is not a good indicator of complexity
4. The measure is a good indicator of complexity, but of little help in understanding how to improve code
5. Tool support is unsatisfactory, particularly in minimizing the spent time on the measurement and facilitating an understanding of the measurement output.

Considering these reflections, we can conclude that not only are measures potentially unhelpful, but also that company regulations and non-optimal tools thwart the full adoption of measures.

The modes of responses in Table 14 show that the first four measures in the table are the least known. Nearly two-thirds of respondents did not know about the first three measures. Similarly, although the last five measures of the table were known by most respondents, they have never been used systematically. Besides the measures that we suggested, respondents also mentioned several measures that they had used; however these were either alternatives of size measures (e.g., number of methods) or measures unrelated to complexity.

Table 14 Measures and their use represented in three categories

	Regularly used	Not used	Never heard
Chidamber and Kemerer	1	33	66
Halstead	2	33	65
Henry and Kafura	1	35	64
Readability measures	6	45	49
McCabe	11	48	41
Fan-in	7	66	27
Fan-out	10	64	26
Change	22	70	8
Size	36	56	8

4.5 Influence of Complexity on Maintenance Time

Understanding the influence of complexity on maintenance time is necessary in order to make decisions on conducting complexity management activities. If complexity has a relatively small influence on maintenance time, it would be difficult to decide whether it is worth spending effort on complexity reduction.

The results in this section aim to increase understanding of the complexity influence on maintenance time. They are inconclusive, however, as the estimates are based on educated guesses rather than quantitative assessment methods. *It is true that such an estimate is subjective, and cannot be used as is. It value, however, is that it provides an insight into the scale of complexity influence. Does complexity increase maintenance time by 10–20%, or 60–80%, or two-fold, or multi-fold or another order of magnitude?*

Figure 13 presents the results of the influence of code complexity on maintenance time of code. The statistical mode of the estimates is 27% corresponding to 250–500%. Twenty seven respondents believed that complexity roughly increases maintenance time by a factor of 2.5–5 times. Generally, 62% of respondents believed that complex code takes more than twice as much effort as maintenance compared with simple code. In fact, only seven respondents thought that the code complexity has insignificant influence on maintenance time. This result means that complexity management activities are necessary because a significant reduction in complexity promises to decrease the maintenance time multiple times. The respondents also commented on how they had estimated complexity influence on maintenance time. Four stated that they remembered some examples of simple code and complex code that they had modified in their practice. They remembered roughly how much time code modification took and made general estimates. One respondent noticed that in her/his

experience, complex code (usually defect-prone) took a multi-fold longer time to correct defects than modifying the given code. One respondent stated that the estimation was a pure speculation. Two respondents found it difficult to make such an estimate. Generally, the estimates indicate that there is a high likelihood that complexity increases maintenance time by multiple times.

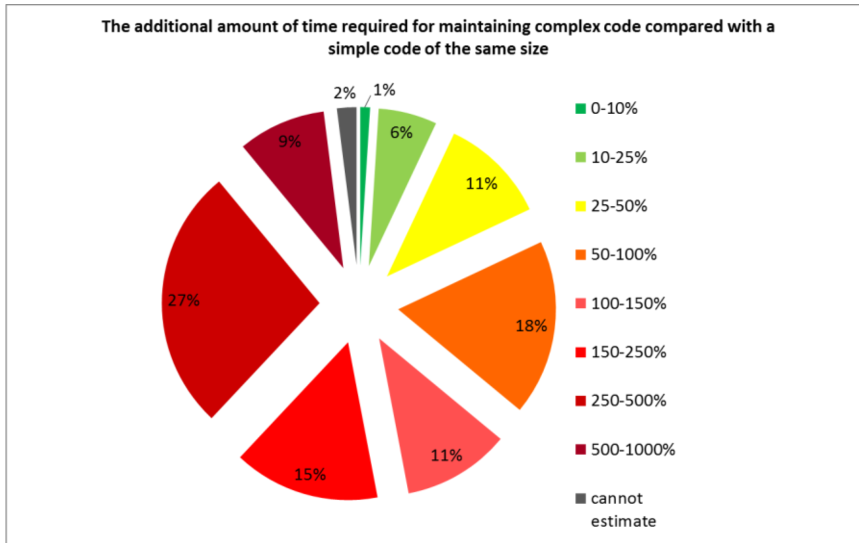


Figure 13 Influence of complexity on maintenance time of code

4.6 Cross-Sectional Data Analysis Results

In the previous five subsections, we presented the five main groups of results of this paper. Here, we investigate whether the demographic data significantly affect the results presented so far. These data correspond to the four pieces of statistical analyses described in Section 3.6.

4.6.1 Type of job and assessment of code characteristics

This section presents results on whether the assessment results of code characteristics are associated with type of job. Table 15 presents the code characteristics and corresponding p and Chi-Square values for every characteristic. The significance level for p-value is $p<0.05$. P-values for "many operators" (0.014) and "many calls" (0.016) attained statistical significance, indicating that there is indeed a difference between the assessments of "developers" and "non-developers". In both cases, the data suggest that according to the developers' assessment, "many operators" and "many calls" have less influence on complexity increase compared to that of "non-developers". All other p-values are large ($p>0.1$), indicating no significant difference between the assessments of "developers" and "non-developers".

Table 15 Chi-Square test results per code characteristic: type of job and assessment

Name of characteristic	Lack of structure	Nesting depth	Control statements	Misleading Com-Comments	Many developers	Multiple tasks
P-value	0.438	0.679	0.804	0.076	0.317	0.249
Chi-sq.	0.602	0.171	0.062	3.151	1.002	1.327
Name of characteristic	Frequent Changes	Complex Requirements	Many Variables	Many calls	Many operators	
P-value	0.654	0.196	0.472	**0.016**	**0.014**	
Chi-sq.	0.201	1.673	0.518	5.808	6.005	

4.6.2 Respondents' experience and the assessment of code characteristics

The results here show whether the assessment results of code characteristics are associated with respondents' experiences. Table 16 presents code characteristics and corresponding p and Chi-Square values for every characteristic.

Table 16 Chi-Square test results per code characteristic: experience and assessment

Name of characteristic	Lack of structure	Nesting depth	Control statements	Misleading Com-Comments	Many developers	Multiple tasks
P-value	NA	0.686	0.213	0.407	0.111	0.040
Chi-sq.	NA	1.164	1.550	0.687	2.538	4.216
Name of characteristic	Frequent changes	Complex requirements	Many variables	Many calls	Many operators	
P-value	0.667	0.460	0.160	0.176	0.659	
Chi-sq.	0.185	0.547	1.971	1.834	0.195	

The p-value for "multiple tasks" is small (0.04), indicating a statistical difference between assessments of "more experienced" and "less experienced" respondents. In this case, the data suggest that according to "more experienced" respondents, the number of "multiple tasks" in a unit of code has more influence in complexity increase compared with the assessment of "less experienced respondents". The rest of the p-values are statistically significant, showing no association between assessment results and respondents' experience. In the case of "lack of structure", one of the values was less than five when calculating the estimated frequencies of its contingency table so it was not possible to conduct a meaningful test.

4.6.3 Type of job and assessment of complexity influence on maintenance time

The results here show whether the assessment results of “complexity influence on maintenance time” is associated with respondents’ “type of job”. The Chi-Square test that was performed based on Table 10 shows a large p-value, p = 0.484 (Chi-Sq. = 1.453), indicating no statistical significance. This means the assessment results of complexity influence on maintenance time are not statistically different across different jobs.

4.6.4 Respondent’ experience and assessment of complexity influence on maintenance time

The results here show whether the assessment results of “complexity influence on maintenance time” is associated with respondents’ “experience”. The Chi-Square test that was performed based on Table 11 shows a large p-value, p = 0.831 (Chi-Sq. = 0.831), indicating no statistical significance. This means the assessment results of complexity influence on maintainability cannot be statistically different due to respondents’ experience.

5 DISCUSSION

Code characteristics as complexity triggers (RQ 1): We have proposed eleven code characteristics in this survey, two of which, *nesting depth* and *lack of structure*, strongly influenced complexity. Compared to other characteristics, these two are usually avoidable because deeply nesting blocks can be averted by using the “return” statement, creating additional function calls, etc. It is also possible to write highly structured code by using meaningful names of function and variables, maintaining line length within good limits, keeping indentations consistent, etc. Other characteristics, such as the number of operators, control statements or function calls, usually cannot be avoided since they are tightly associated with problem complexity.

Our results show that the main two complexity triggers might instead be related to accidental complexity, which can arise due to suboptimal design decisions. Our results also closely relate to a report by Glass [7] that for every 25% increase in problem complexity, there is a 100% increase in complexity of the software solution. A natural question then follows: is it the accidental complexity that quadruples the increased complexity in the solution domain? We believe that there is great value in investing effort to answer this question with a further research because the results of RQ 4 show that complexity has an enormous influence on the maintenance time, which consumes 90% of the total cost of software projects [171].

Figure 10 clearly shows that different complexity triggers (code characteristics) have significantly different levels of influence on complexity increase. This sug-

gests that when creating a complexity measure, the relative differences of such influences should be considered otherwise the complexity measure will miss-estimate the perceived complexity of the given measurement entity. Moreover, when calculating complexity, the weighting for different characteristics can be derived from empirical estimates of code characteristics as complexity drivers. In our case, for example, the *nesting depth* will have a higher coefficient in complexity calculation than the *number of operators*.

The influence of code complexity on internal code quality attributes (RQ 2): The second research question concerns the complexity influence on internal code quality attributes. The results suggest that readability, understandability and modifiability of the code are highly affected by complexity. These results, and those of RQ 1, entail a straightforward conclusion: nested blocks and poorly structured code are the main contributors (at least among the proposed eleven characteristics) in making code hard to read, understand and modify. This conclusion may provide good insight for programmers in order to develop understandable code.

The use of complexity measures in the industry (RQ 3). This part of the survey included only the popular code complexity measures; however, there was an empty field where respondents could register other measures that they used. The results show that all of the measures are used rarely in the collaborating companies, and that respondents have never considered any other complexity measures. There are at least two clear arguments for these results:

1. Either the measures are not satisfactorily good at predicting problem areas,
2. Or the measures are good enough (particularly when used in combination), but software engineers need help in understanding how they can optimally use these measures to locate problem areas and improve the code.

There are also valid perspectives to support both arguments:

1. Designing measures should not be based merely on theoretical frameworks because the weighting for different complexity triggers that are considered in complexity measurement can only be derived from empirical data
2. Complexity measures should be evaluated not only for defect prediction, but also for how well they can both locate complex code areas and indicate necessary improvements.

The influence of complexity on maintenance time of the code (RQ 4): If we were to believe the statistical mode of the results then clearly, complexity management can potentially decrease maintenance time by a multiple factor.

Cross-sectional data analysis (RQ 5): The cross-sectional data analysis results support the argument that results obtained for RQ 1–4 of the survey are most likely not associated with respondents' demographics. It was particularly intuitive to believe that certain jobs not largely related to core development activities

would tend to underestimate the complexity effect on maintenance time. Our results, however, show that this is not so, which might imply that practitioners who are not working directly with software design are, nevertheless, well aware of the complexity effect on maintenance time.

Future work: We are planning two further studies to directly follow this study; specifically, we will:

1. Circulate the survey to a wider range of software developers, including the open source community, to gather results from a wider arena of products and development paradigms, and
2. Design a complexity measure that takes into consideration the assessed influences of code characteristics.

6 VALIDITY THREATS

Notably, when analyzing the results obtained on code characteristics as complexity triggers, these results are limited to the eleven characteristics proposed in this study, which creates a construct validity threat. If more code characteristics had been used in the study, the influence of characteristics on complexity would differ in Figure 10. For example, if we had added more characteristics (e.g., "inheritance level" and "usage of macro-code") to the survey, the number of the most influential characteristics might have increased. This means that "nesting depth" and "the lack of structure" might not be the only important characteristics to consider in coding. This should be considered when applying these results in practice. Nevertheless, adding more characteristics will not change the estimated influence of code characteristics, which means that *nesting depth* and *lack of structure* remain very influential characteristics.

There is also a possibility that several corporate respondents had worked in the same organization/team. A common practice in software development organizations is to decide the standard tools to be used by the organization. Using software measures also complies with this practice. Therefore, if five respondents from the same organization answered the survey, they might all indicate that they use the same measure. Whilst this does not mean that this measure is used more often than others, it does mean that in a particular organization the given measure is adopted for regular use. By including seven companies (including several organizations within each) and two universities in this study, this threat has been significantly minimized. Nevertheless, employing a wider range of companies or domains in this survey would likely result in a markedly more accurate picture of the use of measures. It would be particularly interesting to determine those measures used in open source product development because there the use of measurement tools is fundamentally regulated in a different way. While tool choice is often affected by corporate regulations and standards [172], open source developers are more likely to have greater freedom in their choice of tools.

Another construct validity threat arises due to the possibility that respondents did not actually understand the measures investigated in the survey. It is possible that respondents use a tool that shows values of complexity using a certain measure, yet despite using these values, they still do not know the name of the measure. Thus, when encountering this measure in the survey, they might have marked it as "have not heard of". In the survey, we have partially mitigated this validity threat by providing explanatory text on what a given measure actually shows. It may well be the case that even these explanations do not shed light on whether the given measure was actually known, although this is unlikely.

The four internal quality attributes of code in Section 3.3 were chosen based on two important points. Firstly, the attributes should be simple and direct to enable respondents to make a clear logical connection between them and a complexity otherwise a validity threat of misinterpreting the attribute and the entire question could occur. For example, if we used *conciseness*, respondents might have difficulty in understanding what "conciseness of code" is and thus might provide a flawed answer. Secondly, as we are interested in internal quality attributes that directly affect developers' work on maintainability, we did not want to expand the survey to explore the effect of complexity on any quality attribute in particular.

We designed even-point, Likert scale questions to avoid mid-point values. We argue that mid-point values should not be used because some respondents might opt for them if the question is perceived as difficult and requires more thought. The survey questions did not imply the necessity of mid-point values so we believe that the six-point scale was adequate.

Two factors can cause a construct validity threat when estimating the influence of complexity on maintenance time (RQ 5). The first factor concerns the interpretation of what is simple code and what is complex code. We suggested comparing the maintenance time spent on simple code with that spent on complex code. Since respondents could have their own interpretations of *complex code* and *simple code* in our survey (RQ 5), such a comparison is based on a purely subjective interpretation of the definition of complex/simple code. The second factor concerns the estimation itself, which is neither quantified in any way nor derived from a specified mechanism that used by respondents. These results are derived only from what respondents believe based on their experience and knowledge so we acknowledge that these results should be used cautiously when making inferences or predictions.

The classification of *developers* and *non-developers* for the cross-sectional data analysis might not have been an optimal choice because the non-developers' group contains several categories of jobs. Unfortunately, we were unable to classify the data based on more categories and conduct meaningful statistical tests due to data scarcity. Therefore, the fact that no statistical significance was attained in this piece of analysis might be due to over-simplification of this category

In conclusion, the assessment of code characteristics and their influence on maintenance time is entirely based on the knowledge of software engineers. While a summary of this knowledge can be valuable, it should not be taken for granted. Evidence based on alternative and more objective measures would be markedly beneficial for this type of study [173].

7 RELATED WORK

A comprehensive list of code characteristics that influence complexity can be found in the work of Tegarden, et al. [174]., who separate code characteristics for several entities, including variables, methods, objects and subsystems. They differentiate nearly 40 distinct code characteristics that can influence complexity differently. They propose that some of these characteristics can be combined as they are similar; however they leave this up to the user of their list to decide on how to do so. Their work is valuable because it provides a comprehensive list of characteristics that can be used to design complexity measures. Gonzalez [175] identifies seven sources of complexity that should be considered when designing complexity measures: control structure, module coupling, algorithm, code nesting level, module cohesion and data structure. Gonzales also distinguishes three domains of complexity: syntactical, functional and computational. Syntactical is the most visible domain, although it can reveal information about the other two domains of complexity.

In addition to the nine measures of complexity in our study, there are also several other measures reported in literature that are more or less as good as for complexity assessment, notably the Chapin [176] complexity measure based on data input and output. Munson and Kohshgoftaar [177] have reported measures of data structure complexity, whilst cohesion measures have been described by Tao and Chen [178] and Yang, et al. [179]. Moha, et al. [180] have designed measures for code smells, where "code smells" can be regarded as an aspect of complexity. Kpodjedo, et al. [181] have proposed a rich set of evolution measures, some of which were considered in our study. Wang and Shao [182],followed by Waweru, et al. [183] proposed complexity measures based on the weighted sum of distinct code characteristics. Earlier, we discussed that weighting can provide a more accurate measure of complexity; however the weighting should not merely be based on the perception of the measure's designer, but on empirical estimates to provide sensibly accurate weights. From this perspective, we believe that our study can provide valuable information for studies that design measures of complexity. Keshavarz, et al. [184] have developed complexity measures, which are based on software requirement specifications and can provide an estimate of complexity without examining existing source code. Al-Hajjaji, et al. [185] have evaluated measures for decision coverage.

Suh and Neamtiu [186] have demonstrated how software measures can be used for proactive management of software complexity. They report, however, that

the measurement values they obtained for existing measures provided inconclusive evidence for refactoring and reducing complexity. They observed many occasions when developers reduced values of complexity measures in the code with no reduction in actual perceived complexity as had been expected. The results of this study support the argument that existing software measures are still far from satisfactory for software engineers when not used in combination with each other.

Salman [187] has defined and used a set of complexity measures for component-oriented software systems. Most of the measures that these introduce are more like size measures (the number of components, functions, etc.). There are also measures similar to fan-in and fan-out, but at the component level. Most importantly, the study shows that complexity has major influence on code maintainability and integrity and that there is lack of empirical data on how existing complexity measures actually perform in industry. Kanellopoulos, et al. [188] have proposed a methodology for code quality evaluation based on the ISO/IEC 9126 standard. This work is distinguished by the fact that they use expert opinions for weighting code measures and attributes for more accurate evaluation of code quality. In two of our previous studies, we have developed measurement systems in Ericsson and Volvo Group Truck Technology [189]. We investigated several complexity measures and chose to use a combination of two measures as a predictor of maintainability and error-proneness. Since we had the close collaboration of a reference group of engineers, we received valuable feedback on how these engineers viewed the introduced complexity measures. One of the most important points they made was that the introduced complexity measures, such as cyclomatic complexity, fan-in, and fan-out, are too simplistic for complexity measurement. According to them, there were stronger characteristics of complexity that needed to be weighed in measurement. This feedback was taken into consideration in the design of this current survey.

8 CONCLUSIONS

Effective complexity management can reduce the maintenance cost and increase the chance of producing defect-free software. Complexity measures, therefore, are developed and utilized as a means of quantification of complexity. Existing complexity measures are developed based on theoretical frameworks, but do not necessarily consider empirical observations of the specific code characteristics that complicate code and how much investment each characteristic has in complexity increase. For these reasons, complexity measurement results often are incongruent with software engineers' perceptions of complexity. In this study, we have conducted a survey to: (i) investigate code characteristics and their contribution to complexity increase; (ii) evaluate how often complexity measures are used in practice; and (iii) evaluate the negative effect of complexity on the internal quality and maintenance time. Our results show that: (i) code complexity has a major influence on internal quality and maintenance time; (ii) the two, top-prioritized characteristics for code complexity are not included in

existing code complexity measures; and (iii) existing code complexity measures are poorly used in practice. This study shows that the discipline concerning code complexity should focus more on designing effective complexity measures; in particular, data from empirical observations of code characteristics as complexity triggers should be used. More work is necessary for a greater understanding of how software engineers can use existing complexity measures for effective complexity management and for the ultimate need of maintainability enhancement.

PAPER 5

Mythical Unit Test Coverage

ABSTRACT

It is a continuous struggle to understand how much a product should be tested before the delivery to the market. Ericsson, as a global software development company, decided to evaluate the adequacy of unit test coverage criterion that they employed for years as a guide for sufficiency of testing. Naturally one can think that if increasing coverage decreases the number of defects significantly, then that coverage measure can be considered a criterion for test sufficiency. To test this hypothesis in practice we investigated the relationship of unit test coverage measures and post-unit-test defects in a large commercial product of Ericsson. Based on the results we would like to indicate that the current unit test coverage measures do not seem to be any tangible help in producing defect-free software.

1 TEST COVERAGE MEASURES

Testing is the process of executing a program with the intent of finding errors [190]. Sufficient testing has a decisive role for product delivery. However, as practice has shown, it is rather a complex task to understand whether a product is sufficiently tested or not. There are several unit test coverage measures which are aimed to quantify the sufficiency of testing. Three popular measures are *statement coverage*, *decision coverage*, and *function coverage:*

> *Statement coverage is the percentage of statements in a file that has been exercised during a test run.*
>
> *Decision coverage is the percentage of decision blocks in a file that has been exercised during a test run.*
>
> *Function coverage is the percentage of all functions in a file that has been exercised during a test run.*

Simply increasing coverage takes effort from software developers, but the deal is that we do not know whether this effort is justified, because we do not know how much the increasing coverage can decrease the number of defects. Several researchers pointed theoretically that simply satisfying coverage criteria can miss important code execution possibilities and leave undetected defects [191, 192]. Other researchers showed tactics, such as *assertions* and *causal analysis*, which can improve defect finding capabilities of tests independent of coverage [193, 194]. In practice, however, the direct effect of test coverage on defect-proneness is still unknown.

2 EXISTING STUDIES

First we conducted a literature survey to find out what research is available on the subject. The survey gave us 29 articles that most likely were related to our study. After a close examination we found that only eight of them have direct relation to the subject of coverage-defect relationship. These papers and their findings are presented in Table 1. It seems that seven out of eight papers support the statement that the coverage measures are weakly correlated with the number of defects. Only paper five in the table presents moderate and strong correlation. Most importantly, in seven out of eight papers one or several of the following issues are present:

1. artificial defects (mutants)
2. uncontrolled confounding factors such as size, change rate, complexity
3. artificial tests and coverage control
4. small products and little amount of defects

These factors reduce the likelihood that the obtained results effectively represent the reality. Essentially only one paper presents data which is sufficiently

close to a practical case [59]. Curiously enough the authors of this paper argue that there is a correlation between defects and coverage, but they do not emphasize the fact that the statistical effect size is very small, which is essential in understanding the adequacy of coverage criterion.

Table 1 Papers and findings on coverage measures

Paper	Context	Summary of Findings
1. [59]	Two large industrial products. Actual defects. Block coverage and arch coverage are measured. cyclomatic complexity and code changes are measured and controlled for	The correlation between coverage and defects is none or very weak. Moreover, the effort required to increase the coverage from a certain level to 100% increases exponentially
2. [195]	Twelve small programs. Actual defects, block coverage, and decision coverage are measured	No association is found between the defects and coverage by qualitative analysis
3. [196]	Interviews are conducted with 605 practitioners to understand whether coverage measures are used as test sufficiency criteria	Mixed responses are obtained. Some use coverage as sufficiency criteria, some others stop testing when they feel the most complex part of the code is tested
4. [197]	Twelve small programs. Artificial defects. Monte-Carlo simulation is used to find out the relationship of defects and coverage. Block coverage and defect coverage are measured	The results do not support the hypothesis of causal dependency between test coverage and number of defects when testing intensity is controlled for
5. [198]	Two large open source products. Actual defects. Test suite size, statement coverage, and decision coverage are measured. Defectiveness is measured as a binary variable. Code coverage is not collected as it is but manually generated and manipulated	Moderate to strong correlation is found between coverage and defectiveness.
6. [199]	Five large open source products. Artificial defects. Statement coverage and (modified) decision coverage are measured. Code size is measured and controlled for	Weak and moderate correlation is found between coverage and defects. Type of the coverage does not have an impact on the results
7. [200]	Experiment on a large software product. Artificial defects. Block coverage and decision coverage are measured. The correlation of coverage and defects is assessed under different testing profiles	Moderate correlation is found between coverage and defects. The correlation is different for different testing profiles.
8. [201]	Experiment on 14 industrial products. Both artificial and actual defects. Tests are generated during the experiment. Decision and condition coverage are used. The size of test suites is controlled for	Coverage measures are weak indicators for test suite adequacy. High coverage does not assume effective test

Having such inconclusive results, we decided to conduct a case study which avoids artificial conditions in the investigated product and controls as many confounding variables as possible in order to obtain more conclusive results for the practitioners.

3 THE INVESTIGATED PRODUCT

The product we studies was a large telecom product developed by Ericsson. The product size was about 2 million lines of code (Loc). The organization consisted of about 150 engineers who deliver several major releases of the product in each year. The organization used mixed Agile/Lean development methodologies, relying on incremental code deliveries by semi-independent development teams. As a frontline development organization they are always eager to identify impediments in their development chain. Thus it was natural to question the adequacy of test coverage measures that were used as recommendations for test sufficiency.

4 METHOD OF INVESTIGATION

We collected all defects per file in a year period of time. Generally, if a file is changed due to a defect correction is tagged as “bug fix” in the corresponding development branch. Therefore, it was possible to count how many files have been defective, and how many defects have been fixed in a file. The defects we measured were usually found during integration and system tests, or were reported by customers. The defects that were found during unit testing were not reported and measured. Oppositely, we measured the two coverage measures per file for unit tests, for the same year as the defects were measured. Since the coverage data was stable over the given year, we only took a snapshot measurement for the given year. Considering that unit tests were done earlier than integration and system tests, we could expect that having high coverage during unit testing would reduce the chance of emerging defects during integration and system testing. The opposite is also true: less coverage in unit testing would lead to more defects in integration and system testing. This means that if there is a tangible negative correlation (<-0.4) between unit test coverage and later found defects per file, then the coverage measure could be further analyzed to understand its adequacy of use. However, if no tangible correlation is found, then the coverage measure can be regarded as an inadequate indicator for test sufficiency.

We also measured size, complexity, and changes of files to understand how they affect the test coverage, defects, and coverage-defect relation. Figure 1 depicts an overview of our analysis.

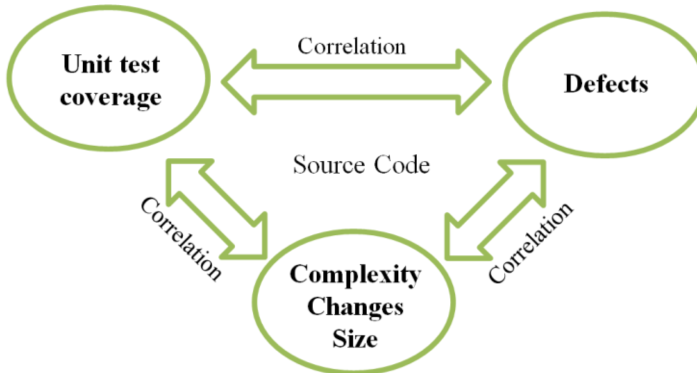


Figure 1 The focus of the study

The original analysis is focused on scrutinizing the relationship of test coverage and defects. However, since both coverage and defects are affected by code properties – complexity, changes, size, – we investigate their influence on the original analysis as well. Probably there are other variables that influence the defects-coverage relationship, such as, developers' experience in coding and testing, the programming language by which the product was developed, integrated development environment which offers testing tools, etc. But we assume that their influence is randomly distributed over source files, so our results will not suffer noticeably.

5 RESULTS

The Pearson/Spearman correlation coefficients between several complexity, size, change measures and defects and coverage measures are presented in Table 2. Important values are boldfaced in the table. The first important thing is that the correlation coefficients between statement (decision, function) coverage and defects are weak (rows two, three, and four under the column of defects). Interestingly the correlation between coverage measures is very strong, indicating that they are very similar to each other (rows three and four in the last column).

This is the reason why the correlation coefficients between coverage measures and defects are nearly the same. Since the decision, statement, and function coverage are strongly correlated, only one of them is used in the correlation analysis with the other variables (the last column of the table). The fact that the correlation between the coverage and defects is weak can be regarded as the first indication of the coverage measures being inappropriate as test sufficiency criteria. However there can be a problem with this kind of conclusion, because the size of files is not controlled for. Our analysis assumed that files with equal coverage should have equal amount of defects. But this assumption is not true, because a file with 1000 Loc and 50% coverage has 500 Loc untested, while another file with 100 Loc and still with 50% coverage have only 50 Loc untest-

ed. You see, the files have equal coverage but there is much bigger likelihood of finding defects in 500 lines of untested code than in 50 lines of untested code. In fact the strong correlation between Loc and defects (0,67/0,53) indicates that bigger size is more likely to defects, and that is quite intuitive.

Table 2 Pearson/Spearman correlation coefficients between change, size, and complexity measures with defect count and coverage of files

N	Property	Measure	Correlation with defects	Cor. with statement coverage
1	Defect	Defects	1	-0,19 / -0,13
2	Coverage	Statement coverage	**-0,19 / -0,13**	1
3	Coverage	Decision coverage	**-0,19 / -0,13**	**0,91 / 0,87**
4	Coverage	Function coverage	**-0,18 / -0,14**	**0,87 / 0,86**
5	Change	Versions	**0,79 / 0,62**	-0,14 / -0,07
6	Change	Developers count	0,76 / 0,63	-0,14 / -0,06
7	Change	Changed code	0,61 / 0,55	-0,11 / -0,08
8	Change	Added code	0,58 / 0,53	-0,11 / 0
9	Change	Deleted code	0,51 / 0,55	-0,1 / -0,08
10	Change	Age	0,31 / 0,27	-0,27 / -0,21
11	Size	Statements	0,62 / 0,49	-0,17 / -0,11
12	Size	Loc	**0,67 / 0,53**	-0,18 / -0,12
13	Complexity	Cyclomatic complexity	0,64 / 0,48	-0,18 / -0,12
14	Complexity	Maximum block depth	**0,42 / 0,42**	**-0,4 / -0,33**
15	Complexity	Parameter count	**0,52 / 0,45**	0 / 0
16	Complexity	Percent comments	**0 / 0**	0 / 0
17	Coverage density	Statement cov./Loc	**-0,06 / -0,25**	–
18	Coverage density	Statement cov./Versions	**-0,18 / -0,3**	–

The same problem emerges for files with different change rates: files that have more changes (versions), have been under more intensive development, and therefore are more prone to have defects. These problems of size and changes create validity threats for our analysis results. Therefore, in order to neutralize the effects of size and changes on defect-coverage analysis results we created two additional measures – average coverage per Loc (statement coverage over Loc) and average coverage per version (statement coverage over versions). Lines 13 and 14 in Table 2 show that the Spearman correlation coefficients between the new measures and defects are a little improved, (-0,25 and -0,3), however they still did not gain any tangible value. At this point we reached to an important conclusion for this study: Increasing coverage does not necessarily decrease the amount of defects.

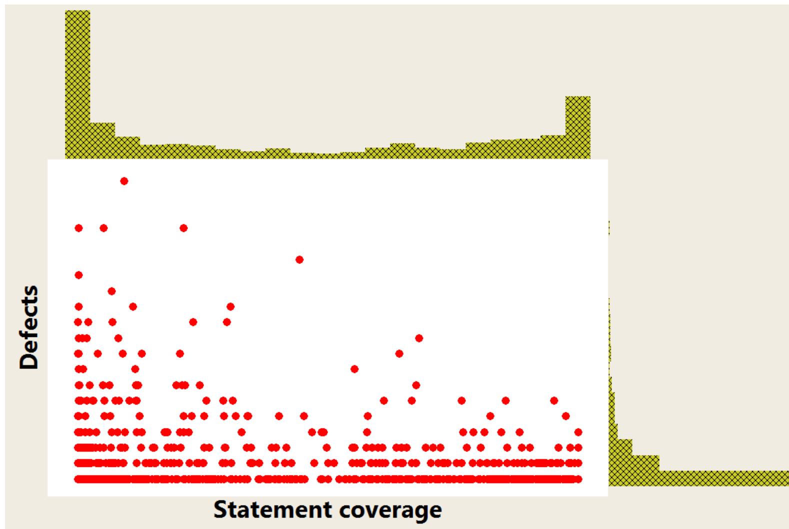


Figure 2 Marginal plot for statement coverage and defects

The only thing we can say is that increasing coverage creates a slight tendency of decreasing defects. This fact is illustrated in Figure 2. Even though the figure shows a downward association between defects and coverage for some files (dots), for most of the files, which are closer to the origin of the coordinating system, this association does not exist. We also found that besides the maximum block depth (max block depth), the rest of the complexity and change measures presented in Table 2 were strongly correlated with either Loc or Versions, so they did not have any additional impact on the coverage-defect relationship, when Loc and Versions were already controlled for.

6 The Effect of Complexity

As we mentioned earlier, there was only one measure which was not correlated with the size and change measures strongly, and this measure was max block depth. This measure shows the maximum level of nesting in a file. Table 2 shows that max block depth is the only measure which has tangible negative correlation with coverage (-0,4/-0,33), suggesting that it might be hard to write tests for nested code. At the same time max block depth has tangible correlation with defects (0,42), indicating that nested code might be more prone to defects than simpler code. Since max block depth is correlated both with defects and coverage, we found it interesting to understand defect-coverage relation in the context of max block depth. Figure 3 shows two contour plots. The upper plot shows the relation of statement coverage, max block depth, and defects. The dark green area indicates no defects, while areas towards yellow and then red indicate increasing number of defects. What is very interesting about the plot is that all of the defected areas are situated in the right-hand side of the plot, clearly indicating that defects emerge only in places where the level of nesting gets

higher. However, we cannot draw the same conclusion for the coverage-defect relationship, since the defects are along the line of coverage axis. It is true that there is more red area in the bottom of the plot but still upper part contains red and yellow areas as well. It is important to recall that coverage itself was negatively correlated with max block depth, so some of the files in the bottom-right red area have low coverage right because of high nesting level. This essentially means that nesting has double effect on code: it both increases the defect-proneness of code and complicates the process of writing tests. It is worth to notice that max block depth is not a solidly defined measure for source files as entities. It only indicates nesting level for a block but not for a whole file, but still we observe its effect on file level.

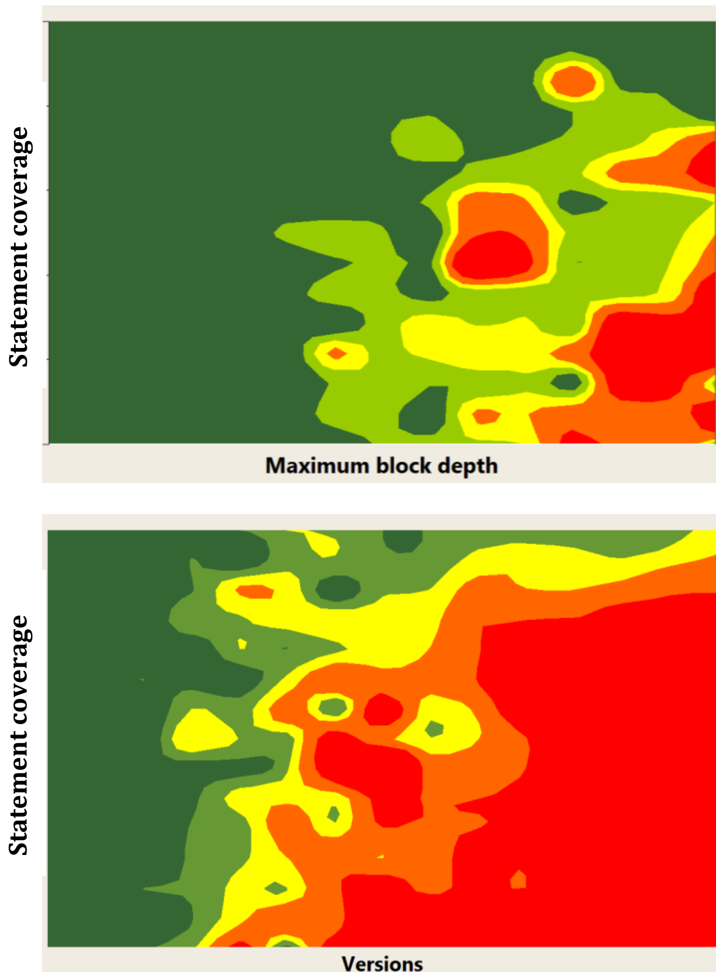


Figure 3 Contour plots of statement coverage and defects

A more adequate complexity measure based on nesting might indicate much larger effect on the coverage and defects. The bottom plot shows a similar presentation of the relationships between statement coverage, defects, and versions. This plot also clearly shows that files which have little amount of versions (less changes) did not have defects, no matter they were tested or not. When we substituted the versions by Loc in the second plot, we got a very similar picture. Thus, the areas of code that are small in size or are not changed are defect-free, and this is quite natural by the laws of probability, because it is more likely to find defects in larger files or in files that are under intensive development.

Generally speaking, in the first plot we see a relation of code complexity, coverage, and defects, while in the second plot we see a relation of development size, coverage, and defects. Usually complexity is more manageable than size and changes. The latter ones are not always possible to reduce since they are the core constituents of product development and delivered functionality: no value can be delivered without developing new piece of code or modifying an existing one. Meanwhile certain amount of complexity can be reduced by contrivance and smart coding tricks, and therefore it is possible to partially control the effect of complexity on defects and coverage.

7 CONCLUDING REMARKS

Decision coverage, statement coverage, and function coverage are popular measures that purport to indicate test sufficiency. However, 100% coverage does not entail 100% tested code. Moreover, the results of this case study suggest that: 1) the adequacy of unit test coverage criterion in Ericsson was a myth, 2) it is well worth to conduct similar case studies in other domains in order to understand how general this problem is in practice, 3) international standards such as ISO 26262, IEC 61508, ANSI/IEEE 1008-1987, and DO 178B need to reconsider their recommendations of unit test coverage measures as criteria for test sufficiency, 4) developers should realize that managing complexity both decreases error-proneness and facilitates testing of code.

PAPER 6

Validating Software Measures Using Action Research

ABSTRACT

Validating software measures for using them in practice is a challenging task. Usually more than one complementary validation methods are applied for rigorously validating software measures: Theoretical methods help with defining the measures with expected properties and empirical methods help with evaluating the predictive power of measures. Despite the variety of these methods there still remain cases when the validation of measures is difficult. Particularly when the response variables of interest are not accurately measurable and the practical context cannot be reduced to an experimental setup the abovementioned methods are not effective. In this paper we present a complementary empirical method for validating measures. The method relies on action research principles and is meant to be used in combination with theoretical validation methods. The industrial experiences documented in this paper show that in many practical cases the method is effective.

1 INTRODUCTION

A measure is considered valid if it fulfills the theoretically required properties and has some kind of predictive capability of a variable of interest. Validity of software measures is essential for designing and applying assessment tools for software development organizations. The use of rigorously validated measures provides valuable information on the developed software and development processes. Software developers use this information to obtain valuable knowledge on the quality, risks, opportunities, productivity, cost, and speed associated with the software development product and process. A well-established opinion is that a measure should be designed based on a firm theoretical ground and be tested many times in practice in order to be considered valid [27]. There is much work done for providing theoretical methods for validation: Kitchenham, et al. [158] introduced a framework for software measurement validation based on a set of predefined rules. Schneidewind [157] proposed a methodology for measurement validation based on a set of validity criteria. Briand, et al. [23] introduced property based measurement in order to facilitate the measurement design and validity check. Issues concerned with difficulties of having a generally accepted measurement validation framework is discussed by Sellami and Abran [202] to a great detail.

Empirical validation, however, seems to be a more difficult task: in order for a measure to be empirically valid, it is expected to have some kind of predictive power [27]. We continuously see studies using statistical models for assessing the predictive power of measures. Examples are [97, 98, 203, 204]. However, in practice, despite the extensive use of measures and existed validation frameworks, there are many measures which are simplistic for what they are designed to measure. Moreover, we continuously see many scientific reports still trying to validate such old and established measures as cyclomatic complexity [106] or Halstead measures [165]. These are the symptoms of not having a well-established and generally-agreed-upon framework for software measurement validation. Surely statistical models are powerful for examining whether a given measure predicts the intended variable of interest of software system. But, how can we conduct a sample measurement on this variable, so we can make a statistical model for validation? The practice shows that such a measurement is very difficult. Many papers use the defect count as a measure for quality and validate various measures against it [28, 205]. Nevertheless, they rarely question how accurately the defect count can be done (not so accurately in many cases as we show in this paper) and to what extend the defect count represents the quality. Problems similar to this one are ubiquitous in software measurement. These can be related to assessing code maintain-ability and error-proneness, requirements understandability, models' complexity, development speed, etc.

When the variable of interest is measurable, at least for historical or a subset of data, statistical models are often used for validating measures. When the variable of interest is not measurable directly, experimental methods may be used to obtain some kind of ranks or values for the variable of interest. Then these

ranks and values can be used in statistical models for validating the given measure. In many cases, when the variable of interest is not directly measurable, and the organizational context cannot be reduced to an experimental setup [143], it is difficult to conduct empirical validation. To the best of our knowledge, there are no validation approaches which would help to empirically validate software measures when the variable of prediction is not accurately measurable. In this paper we illustrate how the action research can be used for empirical validation of measures when the variable of interest is not accurately measurable. The research question that we address in this paper is:

> *How can we validate software measures when the prediction variables are not accurately measurable and when the organizational context cannot be reduced to an experimental setup?*

The results of this paper are derived from data that we collected from a set of action research projects in five large software development companies, where we previously designed and evaluated a variety of internal quality measures. Action research is a methodology which permits the application of a designed method directly in the area of its intended use, thus allowing collecting valuable feedback and refining the method accordingly. The results show that action research methodology is suitable for conducting empirical validation of measures. Particularly it helps to understand the effectiveness of a measure in what it is designed for assessing. Action research also permits evaluating the improvement possibilities of a measure. We argue that the effectiveness of a measure should be evaluated by the help of software engineers who use that measure. The repetition of such evaluation with many software engineers and in many software development organizations helps building up solid knowledge about whether or not the measure can be ultimately accepted for adoption and regular use.

2 A Recap of Measurement Validation Research in Software Engineering

There are two complementary groups of measurement validation methods in the literature, theoretical and empirical [24]. The first methods of the first group usually define properties or rules that a given measure should fulfill in order to be regarded a correct measure of the given attribute. The methods of the second group are used to find out whether the given measure has the desired predictive power for predicting the variable of interest. Table 1 presents examples of validation methods, requirements for measure's validity, and an example measure. The fourth column of the table presents the validation method proposed in this paper. In the coming two subsections we discuss the current state of theoretical validation methods and empirical validation methods based on statistical models.

2.1 Theoretical Validation

Theoretical validation is needed for understanding the properties that a measure should fulfill and the rules that it should comply with, indicating what that measure should be and what it should not be. The theoretical validation helps to understand whether the measure is actually a measure of the intended measurement attribute of software entities. Measurement attributes are *size, complexity, cohesion, length, coupling, change frequency,* etc. Entities are *code, requirements, test cases, architecture, development processes,* etc. Not knowing what attribute we actually measure by a given measure makes the use of that measure difficult, as we cannot correctly understand in what prediction (evaluation) mechanism it should be applied for.

The properties and rules, in their turn, are derived from the essential understanding of software attributes. For example the *size* attribute should have additivity property because the essence of the size concept (attribute) indicates the amount of something: when adding more of this something it should have more size then earlier. Thus when designing a size measure, it should comply with the determined additivity property. In this manner properties of the other attributes are determined in order to ease the design of measures. One of the early works dedicated to software measurement validity is reported by Weyuker [156]. In this work she formulates a set of properties which shall be necessary for newly defined complexity measures. Even though the work was criticized for not providing a complete set of properties it provided a fresh ground for measurement validation frameworks.

Table 1 Techniques of validating measures

	Theoretical validation	**Empirical validation**	
	All cases	Response variable is measurable	Response variable is not measurable
Validation method	Briand, et al. [23] framework	Statistical models	Action research, reference group
Requirement for validity	A measure is valid if it fulfills the required 5 properties of complexity	A measure is valid if it predicts a variable of interest	A measure is valid if it is a good indicator of a variable of interest (qualitative)
Example measure	McCabe's complexity (M)	M	M
A valid measure of complexity if	it fulfills 5 properties of complexity	it predicts the number of defects	it affects the readability of the code

This was called a property based measurement [23], the essence of which was to understand what basic properties a particular measure shall fulfill in order to be a valid measure. Another framework for validating measures is reported by Schneidewind [157]. The methodology relies on six validity criteria for a measure: association, consistency, discriminative power, tracking, predictability, and repeatability. The author claims that fulfilling these criteria provides a good rationale for considering a measure valid. Briand, et al. [23] presented property based measurement for facilitating the selection and validation of measures. Their method is based on the idea that every measure should fulfill a set of predefined properties in order to be qualified as valid. These properties are based on human perception, relying on the intuition. For example we know that the size must be additive because its essence and definition dictates so. Thus any size measures should fulfill the additivity property. In another study Briand, et al. [27] observe that it is very hard to agree upon a general rule or method for measures' validation. They notice that a general validation framework cannot be expected from one researcher or from one study, it is rather repetitions and replications of multiple studies that builds trust on a method. Kaner [162] explores the use of software measures in the field of software engineering and found that there are too many simplistic measures that do not measure whatever they purport to measure. He notices that the use of such measures is not rare, so it is better to put more effort in the design of measures to get more meaningful data.

Having theoretical frameworks for measures' design and validity, researchers in the field examined how applicable the measurement theory is in practice. For example Briand, et al. [206] found that the application of software measurement theory sometimes can be questionable due to several factors, such as undefined scale types of several measures, endless discussions of what exact properties complexity measures should fulfill, what kind of statistical model should be used for measures' validation, etc. In a mapping study Kitchenham [14] concludes that there is a large body of empirical validation of measures, however it seems in measurement research we do not quite know how much a measure should be validated, so even some old and well-established measures can be still validated by researchers. Her findings show that only statistical models for finding the relationship between measures and defects possibly are not enough for thorough validation of measures. Our study supports her observation as even measuring the number of defects can be quite tricky task. A recent study reported by Mair and Shepperd [207] found that software engineers' participation for designing good measures and statistical models is crucial, however this consideration is often ignored. This study also concludes that software engineers' participation should be considered when designing prediction measures in companies. Meneely, et al. [208] conducted a systematic literature review on software measurement validity to understand the main problems and current state of validation methods. They found 47 validation criteria for measures reported by 20 authors. The authors concluded that several authors completely disagreed on a number of validation criteria, which shows the non-profound nature of those criteria. McGarry [209] (p. 128) emphasizes the importance of users feedback

for designing adequate measures, however he does not specify an understructure by which the communication should be established for getting feedback and refining the measures. Elbaum and Munson [210] investigated the relationship of software measures and defects in an empirical study. Among other findings they report that it is extremely difficult to accurately measure the number of faults for a given entity of code (module, file, or function). They mentioned nearly the same reasons of problems for this measurement as we presented in this paper. This study also comes to support our observation that there are alternative views on how we should count a given measure, even if all the alternatives are theoretically valid. Sellami and Abran [202] found that the existing validation frameworks rely on validation criteria of different philosophies. As a way forward they suggested building consolidated framework based on multiple validation types. There are also international standards of software measurement, such as ISO/IEC 15939 [211] and ISO/IEC 25000 [212]. These standards aim to facilitate the design and evaluation of measurement and also provide a common vocabulary to the community. However, they still need consolidation for providing explicit guidance for measurement validation. El-Emam [213] discusses the importance of both theoretical and empirical validation and emphasizes the importance of distinguishing between internal and external attributes of software products. Then he acknowledges that usually it is possible to design measures for internal attributes such as complexity or coupling, which are used to assess the external attributes such as maintainability or error-proneness.

2.2 Validation Using Statistical Models

Statistical models (a variety of regression models, Bayesian networks, Markov models, Neural networks, etc.) are widely applied to understand whether measures of internal attributes of software can predict external attributes. Internal attributes, such as complexity or change rate, are not direct representatives of external attributes such as software quality, risks, or development speed, which are the actual variables of interest of software developers. Internal attributes rather influence external attributes and their main merit is to predict the external attributes. This prediction mechanism is easily developed by statistical models, when there are accurate measures for both internal and external attributes. However, as practice shows, designing accurate measures is rather challenging and sometimes problematic. Today, there are many papers using the defect count as the measure of external quality, and then validate different measures by versatile statistical models against the defect count [205]. We should not only notice that there are other much important aspects of external quality, such as, maintainability, but also that the defect count is sometimes highly inaccurate and inadequate (we argue this claim in subsection 6.1.4 by concrete examples). When the aim of a statistical model is to predict the number of defects for the sake of planning or resource allocation, its use is justified. However, if an organization wants to have more in depth analysis to identify product areas of bad quality these models become inadequate. The reason is

that the number of defects is a narrow characteristic of quality. Moreover, it is the symptom of deeper multifactor problems such as accumulated technical debt, inadequate development processes, size of the product, and other organizational factors. Hall, et al. [205] conducted a systematic literature review in the area of software defect prediction and found that nearly all statistical models, which perform well in the original context, failed to perform well when applying in different organizations or products. Most of the models that are good in prediction are trained in their context of use. Moreover, the measures used in these models were not consistently good or bad predictors across the studies, and sometimes even had contradictory results. This is a strong sign that the defect count is not a clear representation of problem areas but rather a manifestation of multifactor problems. For this reason it is not always adequate to validate a measure by assessing its defect prediction capability.

When the variable of interest is not directly measurable, experimentation can be applied for acquiring measures from human assessors. Such variables are maintainability, ease of integration, readability, etc. In this case the human assessors might be requested to analyze the given set of artifacts and give some ranks of maintainability, readability, etc. Then these ranks (as measures) can be applied for designing prediction models. Experimentations on software measurement are expected to be conducted in controlled conditions. However, in order to achieve controlled experiments in software engineering, the experimental setup needs to be simplified considerably. This simplification usually affects the experimental system to an extent that it becomes no longer a representative of a real-word system. For this reason, often the results obtained from such experiments are not applicable for software development organizations [143]. In a survey on experimentations in software engineering Sjøberg, et al. [214] reported that 103 studies were conducted between 1993 and 2002. Among these 103 studies there was only one study in the field of software measurement. In fact, most of the studies were done in the areas of comparing domain specific languages and code inspection techniques. These areas are the least affected by organizational process and therefore are easier for isolated controlled experiments. Oppositely, there were very few experiments on such subjects as productivity, cost, and risk assessment (one study per each). This kind of scarcity of the data is due to the difficulty of conducting meaningful experiments in the field of software measurement.

3 A Method for Validating Software Measures

In majority of cases it is not possible to design accurate measures for external product attributes, so the validation of internal attributes cannot be conducted by help of statistical models. In such cases we argue that action research can be successfully applied for validation. There are two factors that make action research powerful in validating measures. First, it permits the validation directly in the application area. Second, it relies on systematic define-refine-redefine typical action research process with practitioners, which allows ultimately

shape the intended measure and either accept or reject it for further application. In this process, the qualitative feedback of practitioners, who on practical examples evaluate to which extent a measure has influence on the variable of interest, has a pivotal role.

Figure 1 shows the evaluation method as an action research cycle. It is aligned with the action research methodology, and is adopted for validating measures.

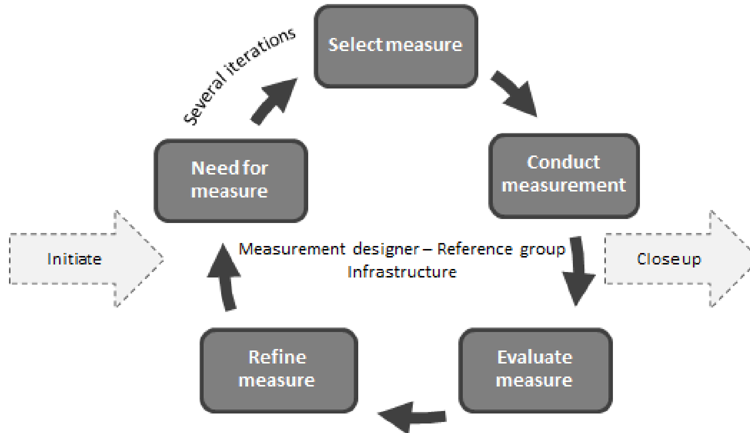


Figure 1 Action research cycle for validating measures

The measures are selected (defined), calculated, evaluated, and redefined based on the evaluation until they are perceived to be good measures. In the middle of this cycle is the 'measurement designer' – 'reference group' infrastructure which jointly refines the measure, until it reaches to an adequate condition for use. Initially the organization forms a "measurement designer" – "reference group" infrastructure. On the one hand, the measurement designer is the person, who is assigned for designing and validating measures for the organization. This can be an external or internal researcher, a measurement expert, a design architect or other knowledgeable person in the field of measurement. On the other hand, the reference group is a group of practitioners who work closely with the artifacts that are to be measured. The members of the reference group should have deep insights on the measurement artifacts and directly work with these artifacts on a systematic basis (developers, testers, architects). The measurement designer sets up systematic meetings with the reference group. The steps shown in Figure 1 are:

5. The measurement designer decides on designing or using a measure for predicting a variable of interest. Variables of interest are not accurately measurable, such as maintainability, modifiability, ease of integration, etc.

6. The measurement designer, with the reference group, decides upon the measurement entity and the measure that is to be validated. Measurement entities can be source code files, functions, requirements, models, test cases, etc. The measures usually are designed for known software attributes such as complexity, size, length, cohesion, coupling, change frequency, etc.
7. The measurement designer conducts measurement by developing or using available automated tools. She/he structures the measurements results in a file (files), where the name, location, measurement result, and other necessary information per measurement entity are available
8. The measurement designer evaluates the measures against the reference group's understanding of the non-measureable property: She selects a sample set of measurement entities in such a way, that for half of them the measurement values are big and for the other half the measurement values are small. The sample set is selected considering the size of the population, which is usually known. Afterwards, the measurement designer presents the selected set of entities to the reference group, and they brainstorm together for understanding how effectively the selected measure can assess or predict the variable of interest. They review each measurement entity one by one and share their knowledge on what kind of problems or difficulties they had previously with the given entities in the past development time. The measurement designer writes registers these problems and difficulties per measurement entity in a checklist. The practitioners of reference group brainstorm and find rationale for regarding a given measure effective or not effective. Their agreements and disagreements, and reasons are also registered in the checklist. After brainstorming the measurement designer decides to what degree of accuracy the measure predicts the variable of interest, based on the summarized knowledge in the checklist. The degree of accuracy is decided on a qualitative scale: an example is "bad", "average", and "good". The measurement designer also outlines the enhancement possibilities of the measure, based on the obtained knowledge
9. The measurement designer examines the measurement method for the selected measure: questions that are needed to be addressed for this examination are:
    a. What do we actually calculate by the defined measure?
    b. What inaccuracies are likely to be introduced by the defined method of calculation?
    c. How can we refine the method of calculation so the measurement values can show a better association with the variable of interest?
10. The measurement designer considers the aforementioned questions and the discussions over them with the reference group. She refines

the measurement method and redefines the measure based on acquiring answers for the above questions. Then she plans measurements for a new set of entities. In the next cycle the new set of entities are planned to be reviewed and discussed, and thus more knowledge can be gathered

11. The further process iterates over (Figure 1) until the measurement designer can decide whether the selected measure is validated for final use or not.

We used the essence of the described method in five large software development organizations for validating several sets of measures. In Ericsson we used this method for validating measures of complexity, size, and evolution for source code. In Volvo Group we used this method for validating measures of complexity, coupling, and evolution of textual requirement. In Grundfos and Saab we replicated the validation process for requirements' measures. In Volvo Car Group we used this method for validating measures of Simulink models and textual requirements.

4 AN ILLUSTRATIVE CASE

In this section we present an example of how a measure was validated in practice. The validation process was conducted in a software development organization at Ericsson [140]. We formed the "measurement designer" - "reference group" infrastructure. The measurement designer was a researcher (the first author of this paper) from the Software Engineering Division of University of Gothenburg. The reference group's participants consisted of a measurement team leader, three design architects, an operational architect, and a project manager from Ericsson. We organized biweekly meetings for the evaluation process. The rest of the process complies with the method presented in the previous section:

13. The measurement designer decides on designing a measure for automated identification of risky areas of software code: such code patterns that were error-prone or difficult to maintain. Source code files were chosen as measurement entities. This means that our target was to identify risky source code files.
14. Evolution of files was chosen as a measurement attribute. The measure for validation was the number of check-ins (NR) of files to the version control system in a specified period of time. Participants of the reference group believed that this measure can be effective in identifying risky files when combined with complexity measures. The reasons for such a belief were:

    a. In a specified period of time there were only a limited number of files changed, which means that if we search for error-prone files for a specific period of time, than they must be among the changed ones

    b. Complex files are risky only if they have been changed recently

15. The measurement designer developed a tool (script) and counted the number of check-ins per file for one month interval. The names of files, measurement values, and locations were registered in an excel file.
16. The measurement designer randomly selected 50 files that had big values of NR and 50 files that had small nonzero values. The measurement designer presented the selected set of 100 files to the reference group. The reference group engineers examined the files one by one and discussed their experiences with these files. The measurement designer registered the collected information during the discussion. The results of the collected data were as follows:
    a. There was a group of complex files that changed more often than other groups. Those were usually problematic and were classified as risky
    b. There was a group of files that often changed. Those were header files or simple files that were affected by the changes in other files and had to be changed. These files were not classified as risky
    c. There was a group of complex files that had not changed at all. It was difficult to understand whether in cases of changes these files can be classified as risky, therefore these files were classified as potentially risky and needed a further investigation
    d. There was a group of simple files that do not change. The reference group decided that in practice such files were not problematic and they were classified as not risky with no further investigation needed
17. At this point, the measurement designer concluded that NR is likely to be an effective measure when applied with combination of complexity measures, because it can predict the risky files in:
    a. case a)
    b. the files of case b) can be easily filtered out using complexity measures, files in case d) are filtered out by NR measure, and
    c. case c) needs further investigation by doing additional measurements for different time intervals

For the files that needed a further investigation we decided to add one more action research cycle. The time intervals for additional measurements were chosen to be daily, weekly, and release-wise. Such measurements would permit to understand how complex files behave during different time intervals.

The reference group engineers proposed alternative measures of evolution, such as counting the number of designers making check-ins, and the number of check-ins for rather on a specific development branch than on all the development branches managed by different development teams. As the NR measure showed an initial good qualitative result, the reference group concluded that the measure can be classified as a “good” measure for continuing the validation

process. The measurement designer planned designing tools for conducting measurements for the suggested two additional measures to check whether they outperform the NR measure

After this point the action research cycle repeated. The measurement designer conducted the planned measurements for different time intervals, prepared the files that needed further investigation, and presented to the reference group in the next meeting. He also conducted measurements for the additional two measures. The action research cycle was repeated 6 times (12 weeks) to understand the behavior of different groups of complex files. Continuing similar analysis for 12 weeks the reference group concluded that the NR measure is good enough for being used in combination with the complexity measures. The additional two measures were removed because they were in a strong correlation with NR measure, making these three measures equally effective for the planned use.

5 ORGANIZATIONAL CONTEXT OF THIS EXPERIENCE REPORT

Prior to this study we had conducted several action research projects at five large software development organizations. The collaborating organizations were Ericsson, Volvo Group, Volvo Car Group, Saab AB, and Grundfos. The projects aimed at applying software measures for identifying difficult-to-maintain artifacts for refactoring. In order to get continuous feedback on the ongoing work and determine next steps in the research in each of the company we collaborated with a reference group of engineers. These engineers had extensive knowledge on the developed product and had different positions in the organization. By their help we both designed new measures and used already well-established measures in the literature. There were three different artifacts targeted for measurement in these organizations: textual requirements, Simulink models, and source code of products. Figure 2 shows in how many organizations a given artifact was measured.

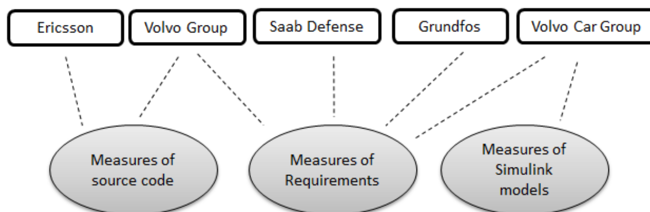


Figure 2 Software artifacts measured in the collaborating companies

A number of measures were used throughout these projects and tested for selection of final use. In the beginning we decided to select measures that are well-established in literature and develop measurement systems based on them.

However when we conducted initial measurements and presented to reference groups for feedback we realized that the measures do not show what we expected them to show. Software engineers' feedback was that their perception is not aligned with what measures show. Besides they had also diverse opinions on what should be measured. In order to overcome the chaotic situation we decided to collect all seemingly relevant measures and start a validation process by the help of reference group engineers. Conducting this process in different organization we not only designed the targeted measures for them but collected knowledge on how action research can be used for validating software measures. This collective knowledge is then methodized in this paper as a step forward for empowering the sector of software measurement validation.

6 RESULTS FROM VALIDATING MEASURES IN COMPANIES

In this section we report the results that we got when validating various measures in the collaborating companies. We show how certain measures were demonstrated to be poor ones and other measures to be good ones for the initially defined purposes.

6.1 Measures of Source Code

Prior to this study the authors of this paper designed measurement systems for Ericsson and Volvo Group. The measurement systems used a combination of several measures for identifying such areas of code that are error-prone or difficult-to-maintain. The measures that we used, was validated by Action research. In the coming subsection we discuss results of validation of them in the collaborating companies.

6.1.1 Size

In practice it is quite common that size measures are used as quality estimators. For example the number of source lines of code and different measurement variations of them. First of all we ought to mention that size measures are quite effective in effort estimation [215]. However it is misleading to use size measures for assessing quality for the need of any type of improvements. Why is it so? Let us assume that we have 2 source code files and we would like to assess their quality for improvements. Suppose one of them has relatively bigger size and historically is reported to be more fault-prone. However, this does not mean that it has worse quality than the other file. If its size is bigger, it most likely provides more functionality, thus delivers more features to the customers, and eventually gains more profit. The size is not only highly correlated with fault-proneness but also with gained revenue, because software of a big size most likely delivers more value. This means that when comparing error-

proneness of code one should consider comparing software modules of equal sizes in order to get a meaningful result. The results of Action research cycles with both reference groups at Ericsson and Volvo Group yielded to a conclusion that size measures are useful for effort estimation and defect prediction for planning, but it must not be used for quality assessment, because any conclusions on quality based on size measures cannot prompt a meaningful action towards quality improvement.

6.1.2 Complexity

One of the most used complexity measures of code is cyclomatic complexity. Also there are some controversies about this measure [160, 161] discussing its usefulness, it is still used by many organizations. When T. J. McCabe introduced this measure, he actually never claimed that cyclomatic complexity anyhow purports to measure the complexity of the code as perceived by software engineers. It rather measures the testability of a unit of code. Now we know that in practice it is a poor measure of complexity, however we cannot invalidate this measure against a claim which is never made. For example software engineers at Saab are using this measure as a good practice for testability of code. We investigated this measure at Ericsson and come to a conclusion that cyclomatic complexity does not really indicate the complexity of the code as perceived by software engineers, however it is actually very effective when using it with the combination of evolution measures: we observed that such functions that have big cyclomatic complexity and at the same time has undergone many revision are complex and hard to maintain.

The next complexity measure that we discuss is the number of function calls in a given function. In literature this measure is called *fan-out.* The basic assumption on this measure is that if there are more function calls in a given function then that function becomes more complex. This assumption is quite straightforward from intuition's standpoint, however, it is not clear how this measure shall be calculated. For example, if the same function is called ten times, shall we consider each time as a different call or we shall calculate it as one call, since the number of unique function calls is one? How about calling the same function with different list of parameters each time? Even though all these alternative calculations can be regarded correct, we cannot really know which one is more adequate measure for whatever we are trying to assess. The investigation with reference group engineers at Ericsson and Volvo Group showed that calling the same function many times in the code only slightly increases its complexity, so we used only the unique calls as a more adequate measure.

6.1.3 Evolution

Evolution measures are such measures that show how a given artifact or its attributes change over time of development. Evolution measures are shown to be good predictors of maintainability [108, 216]. When intending to use evolution measures of source code we encountered a problem, which was concerned

with the alternative ways of measures for evolution. For a file, measuring modified lines of code, added and deleted lines of code, the number of check-ins of code designers to the version control system in a specified development branch, the number of check-ins for all branches, the number of code designers that ever opened the given file in a specified time period, etc. are different alternatives of evolution measures. Among all of these measures it is simply not possible to use the most adequate one without using the software engineers' perception of reference group. With their qualitative validation we recorded that the number of designers that make check-ins is a stronger measure for maintainability, however this measure was strongly correlated with the number of check-ins of files so we used the latter one for our measurement system.

Additionally we should acknowledge that measuring the evolution of complexity of source code functions turned out to be quite hard, because:

1. Many of the functions were changing their list of parameters, so it was hard to decide whether they should be regarded as a new function or a reformulation of old ones
2. Many functions were changing their names over time, so we could not track them by automated means

We concluded that due to these two reasons the measurement accuracy is not satisfactory so the evolution of complexity measures should not be used for functions as measurement entities.

6.1.4 Defects

Measuring the number of defects on different software entities has several important roles in software engineering research and practice. This measure is used both for development planning and resource allocating, and also for validating other measures as quality predictors. There are far many papers reported, which use this measure for validating complexity, size, evolution measures, and also complex statistical and analytical models for defect and quality predictions [108, 217, 218]. However, before its use, it is worth to ask how do we measure defects and what does this measure actually show?

In an earlier study we developed method and supporting measurement system for Ericsson for predicting the difficult-to-maintain and defect-prone source files [140]. In order to validate the method initially we decided to use the historical number of defects per file. However, as it turned out, there were several problems with both counting and using this measure. There was no such report as to map defects on files, because it was either very difficult or impossible for developers. Usually the defects are reported per development area. An alternative way of counting defect was to measure which files were changed due to defect correction. This approach had more success, but still there was much noise introduced to the measurement as it was not always clear whether a given file has been changed due to defect correction or other activities. Even if there was a specifically defined branch in version control system for defect correction, all

works done in that branch were not purely concerned with defect correction. This was due to the complex relationships of such activities as defect correction, maintenance, release, and new feature development. Although there was noise in the described measurement, we decided to use this measure for a small set of files and by observing them understand the magnitude of noise. The results showed that for about 20% of the files it is actually not determinable whether a file was changed due to the defect correction or due to another activity. Yet there was even more important reason for being careful with this measure: certain files were in fact changed for defect correction, but they were not the root cause of the defect. We found that there were many cases where the root cause of a defect can be in a complex file, however several other simple files can be affected by that defect and thus undergone defect correction activities. So, when counting defects disregarding the aforementioned issues we might actually underestimate the defect predicting power of complexity measures and overestimate the predicting power of evolution measures (such as the number of changes or revisions).

Lastly, we would like to mention that the criticality of defects also plays a major role when it comes to representing the quality of products by their number of defects. Some defects might take multiple times more effort for correction than other ones, so this will affect the statistical model and the results of their validity. This issue is not well-investigated topic in the field of software measurement, so we do not know its influence on the results we obtain [205].

6.2 Measures of Simulink Models

In Volvo Car Group, where a portion of software development is done by Simulink models, we needed to design measures of complexity to get insights on quality of the models. At the time the company used one measure for complexity, which was the cyclomatic complexity of the code that is generated from Simulink models. We also found several articles which attempt to define measures for Simulink models and measure them [219-222]. One of the tools that was developed by the researchers of Åbo Akademi University can measure several complexity measures, such as the depth of nesting, the fan-in, and fan-out, and the number of in-ports and out-ports of Simulink subsystems. The discussions with reference group engineers of Volvo showed that none of the measures are relevant for measuring the complexity of models for the following reasons:

1. The cyclomatic complexity of the generated code does not anyhow show the complexity of the models from which the code is generated. The code and models are different entities and have different reasons for becoming complex
2. The nesting depth, as originally adopted from code measures, was rather simplicity measure than a complexity measure for models. The reason is that all nested levels in the code are visible in one place making it hard to comprehend for a developer, however, for models, every nested level is isolated piece of implementation, visibly dis-

connected from other parts of the model. So the reference group engineers used it to simplify the models by decomposing it

3. The fan-in and fan-out were defined as the outgoing and incoming calls of subsystems [222]. The investigation showed that these measures in practice do not have any tangibly high values for models. The highest fan-in for all models we examined in Volvo was only two. Thus in practice a subsystem does not call more than two subsystems
4. The in-ports and out-ports of a subsystem was determined by the number of incoming and outgoing signals. Our observations showed that these measures are weak indicators of complexity unless we distinguish two types of them:
    a. signals which are linked to Simulink libraries (strong complexity indicators)
    b. And signals without any linkage (weak complexity indicator)

Based on this evaluation results we started a collaboration with the researches of Åbo Akademi University as to define more advances complexity measures based on the feedback of reference group engineers of Volvo. At the time of writing this paper our collaboration was an ongoing activity.

6.3 Measures of Textual Requirements

In large software development companies, where there are several thousands of textual requirements, automatic quality assessment is much appreciated. We conducted multiple action research projects at Volvo Group, Volvo Car Group, Saab AB, and Grundfos for designing measures of textual requirements, which can help automation of requirements review process. Software measures in the sector of textual requirements are perhaps not as mature as in the sector of coding. For this reason, when trying to find adequate measures, we ended up with collecting a bunch of not evaluated measures for requirements [138, 139, 144, 147]. We started a refinement process with the reference groups and as a result we found that the *number of versions, nesting degree of bulleted text, punctuations, imperative words,* and many *imprecise terms* were weak measures. Instead we found that the *number of conjunctions* and *the number of references to other requirements and modules* in a textual requirement are strong indicators of complexity. Analogous to source code measures the number of revisions turned out a very weak indicator of quality due to two facts:

1. The requirements of higher hierarchy level had more revisions due to containing many requirements inside them. This does not mean that a requirement is frequently revised, but as the engineers of reference group noticed, rather adjusted with its lower members
2. In a long period of development time the requirements got much less revisions than source code (1-2 revision per month), therefore it was impractical to use revision of requirements

Additionally, we should mention that most of the approaches for requirements quality checking that are based on morphological analysis, turned out to be irrelevant for industry. The reason is that the requirements written in companies are not necessarily grammatically assessable. Such examples can be tables of specifications, symbolic representations, pseudocode, etc. Such requirements cannot be grammatically evaluated, because they do not represent “sentences” as linguistic constructs. Moreover, rules, which are designed for morphological analysis, usually enforce on how a requirement should be written, but they do not necessarily reveal the actual problems in requirements.

6.4 Summary of Measures and Validation

Figure 3 shows all the measures that we have validated or invalidated in the five companies. The measures which have cross in front of them were invalidated for the specified use.

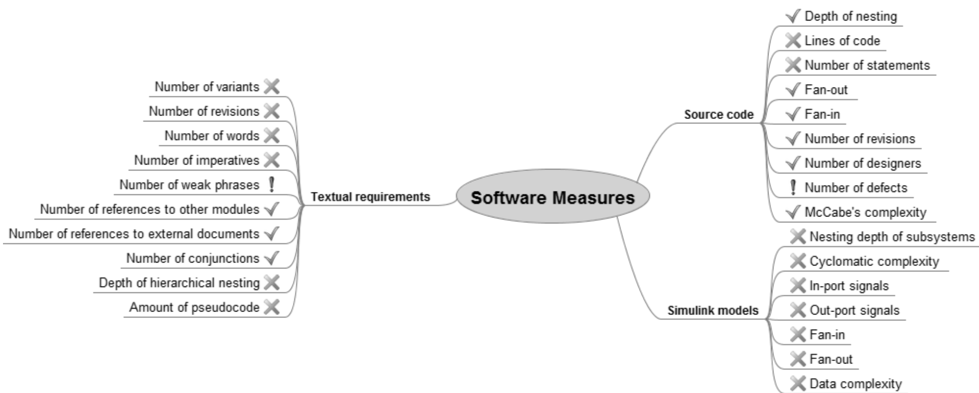


Figure 3 Validated and invalidated software measures - results of our action research projects

The measures with tick in front of them were validated to be either moderate or strong indicators for the specified use. Two of the measures that have the sign of attention are difficult to measure accurately and therefore extra attention was needed.

7 DISCUSSION

The results of validating measures in collaborating companies (section 6) show an important barrier for conducting a meaningful action of validation: what the quantified variable should be, against which a measure can be validated. For example if we want to design a measure for assessing the quality of code then we usually chose the number of defects as a variable against which the measure can be validated. As we indicated in this paper the number of defects itself is very difficult to count, and even more, it is not quite clear what that number

shows. In other cases there is even no clearly defined variable for validating software measures. For example if we would like to design a good measure for maintainability we cannot know, by any quantitative means, whether the designed measure is a good measure of maintainability. In this case the effectiveness of a measure is determined by how well the measure agrees with what software engineers perceive to be maintainable. However, the software engineers' perception is not quantitatively available by any direct and simple means. Therefore there emerges a necessity of qualitative understanding whether a given measure agrees with software engineers' perception. Yet we can see that many researchers and engineers praise quantitative evaluation of measures more than the qualitative one, even if any quantitative measure is derived from fundamentally qualitative perception. This philosophical confusion sometimes yields to a situation where software engineers get fed up by discussing the possibility of using more advanced measures and return to using old and rather simplistic than simple measures.

As a way forward we suggest using action research cycles and close collaboration with software engineers when validating measures. Careful and stepwise consideration of measures and their systematic refinement during action research cycles can reward with more sophisticated and useful measures, which will provide significantly more insight about the measured artifact. Examples of approaches for creating measures in practical contexts are presented by Iversen, et al. [223] and Moody [224].

Despite the importance of practitioners' participation and collaborative learning process in the measurement design, it is solely the measurement designer who design, refine, and select final measures. The opinions and attitudes of reference group engineers may vary over time, and it is the responsibility of the researcher who collects all the feedback and based on its summary decides which measure should be selected. We would like to remind that if a measure is defined and validated for one type of entity (even though that is a really good measure for that artifact), we cannot just take it and apply for another type of entity without proper consideration.

8 Conclusions

Using valid measures provide valuable insights on developed software and development processes. Software engineers and managers use these insights to make strategic decisions for improving their product. There has been a great amount of endeavor by the research community for providing validation frameworks for software measures. No doubt, the results are encouraging, but there are still open issues to be solved. So far we know that there is no generally-agreed validation framework that can serve for measurement validation of any type.

In this study we suggest that the software engineers should be involved in the process of validating measures. We show how action research principles can be

employed as a complementary step for validating measures. The results reported in this paper show that the involvement of engineers can be a powerful tactic for validating measures. The action research cycles enable to systematically refine and redefine the measures as long as they are not perceived to be satisfactory for industrial use. And when the measures are finally accepted by all parties they are deployed for use. The knowledge that is generated during the action research cycles is organized and preserved as a body of knowledge. We would like to encourage the use of action research for validating software measures, as we believe such knowledge can be gathered and methodized for facilitating the validation of software measures.

REFERENCES

[1] R. N. Charette, "Why software fails," Ieee Spectrum, vol. 42, pp. 42-49, 2005.

[2] R. L. Glass, "Frequently forgotten fundamental facts about software engineering," IEEE software, vol. 18, pp. 112-111, 2001.

[3] J. Bosch and P. Bosch-Sijtsema, "From integration to composition: On the impact of software product lines, global development and ecosystems," Journal of Systems and Software, vol. 83, pp. 67-76, 2010.

[4] F. Brooks, "Essence and Accidents of Software Engineering," IEEE Computer, vol. 20, pp. 10-19, 1987.

[5] T. Mens, "On the complexity of software systems," Computer, vol. 45, pp. 79-81, 2012.

[6] E. Rechtin and M. W. Maier, "The art of systems architecting", CRC Press, 2010.

[7] R. L. Glass, "Sorting out software complexity," Communications of the ACM, vol. 45, pp. 19-21, 2002.

[8] V. Basili, "Qualitative software complexity models: A summary," Tutorial on models and methods for software management and engineering, 1980.

[9] H. Zuse, "Software complexity," NY, USA: Walter de Cruyter, 1991.

[10] T. J. McCabe, "A complexity measure," IEEE Transactions on Software Engineering, pp. 308-320, 1976.

[11] M. H. Halstead, "Elements of software science," Elsevier New York, 1977.

[12] S. Henry and D. Kafura, "Software structure metrics based on information flow," IEEE Transactions on Software Engineering, vol14, pp. 510-518, 1981.

[13] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," IEEE Transactions on Software Engineering, vol. 20, pp. 476-493, 1994.

[14] B. Kitchenham, "What's up with software metrics?–A preliminary mapping study," Journal of Systems and Software, vol. 83, pp. 37-51, 2010.

[15] E. J. Weyuker, "Evaluating software complexity measures," IEEE Transactions on Software Engineering, vol. 14, pp. 1357-1365, 1988.

[16] L. C. Briand, S. Morasca, and V. R. Basili, "Property-based software engineering measurement," IEEE Transactions on Software Engineering, vol. 22, pp. 68-86, 1996.

[17] N. E. Fenton and M. Neil, "A critique of software defect prediction models," IEEE Transactions on Software Engineering, vol. 25, pp. 675-689, 1999.

[18] J. Bosch, "Continuous Software Engineering," Springer, 2014.

[19] J. M. Stecklein, J. Dabney, B. Dick, B. Haskins, R. Lovell, and G. Moroney, "Error cost escalation through the project life cycle," 2004.

[20] B. Edmonds, "What is Complexity?-The philosophy of complexity per se with application to some examples in evolution," The Evolution of Complexity, 1995.

[21] A. Geraci, F. Katki, L. McMonegal, B. Meyer, and H. Porteous, "IEEE Standard Computer Dictionary," A Compilation of IEEE Standard Computer Glossaries. IEEE Std, vol. 610, 1991.

[22] N. Fenton and J. Bieman, "Software metrics: a rigorous and practical approach," CRC Press, 2014.

[23] L. C. Briand, S. Morasca, and V. R. Basili, "Property-based software engineering measurement," IEEE Transactions on Software Engineering, vol. 22, pp. 68-86, 1996.

[24] A. N. Kolmogorov, "On tables of random numbers," Sankhyā: The Indian Journal of Statistics, Series A, pp. 369-376, 1963.

[25] J. Moses, "Complexity and Flexibility," MIT/ESD, 2001.

[26] D. W. Hubbard, "How to measure anything: Finding the value of intangibles in business," John Wiley & Sons, 2014.
[27] L. Briand, K. El Emam, and S. Morasca, "Theoretical and empirical validation of software product measures," International Software Engineering Research Network, Technical Report ISERN-95-03, 1995.
[28] C. Catal, "Software fault prediction: A literature review and current trends," Expert Systems with Applications, vol. 38, pp. 4626-4636, 2011.
[29] N. Fenton, "Software measurement: A necessary scientific basis," IEEE Transactions on software engineering, vol. 20, pp. 199-206, 1994.
[30] N. E. Fenton and M. Neil, "Software metrics: successes, failures and new directions," Journal of Systems and Software, vol. 47, pp. 149-157, 1999.
[31] L. Chen, "Continuous delivery: Huge benefits, but challenges too," IEEE Software, vol. 32, pp. 50-54, 2015.
[32] J. Bosch, "Speed, data, and ecosystems: the future of software engineering," IEEE Software, vol. 33, pp. 82-88, 2016.
[33] B. Fitzgerald and K.-J. Stol, "Continuous software engineering: A roadmap and agenda," Journal of Systems and Software, vol. 123, pp. 176-189, 2017.
[34] J. Humble and D. Farley, "Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation," Pearson Education, 2010.
[35] H. R. Neave, "Deming's 14 points for management: framework for success," The Statistician, pp. 561-570, 1987.
[36] T. S. Bateman and J. M. Crant, "The proactive component of organizational behavior: A measure and correlates," Journal of Organizational Behavior, vol. 14, pp. 103-118, 1993.
[37] R. Weber, "Editor's comments: the rhetoric of positivism versus interpretivism: a personal view," MIS quarterly, pp. iii-xii, 2004.
[38] P. Checkland and S. Holwell, "Action research: its nature and validity," Systemic Practice and Action Research, vol. 11, pp. 9-21, 1998.
[39] R. L. Baskerville and A. T. Wood-Harper, "A critical perspective on action research as a method for information systems research," Journal of Information Technology, 1996.
[40] M. N. Saunders, "Research methods for business students, 5/e," Pearson Education India, 2011.
[41] K. Lewin, "Action research and minority problems," Journal of social issues, vol. 2, pp. 34-46, 1946.
[42] G. I. Susman and R. D. Evered, "An assessment of the scientific merits of action research," Administrative science quarterly, pp. 582-603, 1978.
[43] F. Baum, C. MacDougall, and D. Smith, "Glossary: Participatory action research," Journal of Epidemiology and Community Health (1979-), vol. 60, pp. 854-857, 2006.
[44] D. E. Avison, F. Lau, M. D. Myers, and P. A. Nielsen, "Action research," Communications of the ACM, vol. 42, pp. 94-97, 1999.
[45] R. Baskerville and A. T. Wood-Harper, "Diversity in information systems action research methods," European Journal of Information Systems, vol. 7, pp. 90-107, 1998.
[46] R. Baskerville and M. D. Myers, "Special issue on action research in information systems: Making IS research relevant to practice: Foreword," Mis Quarterly, pp. 329-335, 2004.
[47] L. Mathiassen, "Collaborative practice research," Information Technology & People, vol. 15, pp. 321-345, 2002.

[48] A. Sandberg, L. Pareto, and T. Arts, "Agile collaborative research: Action principles for industry-academia collaboration," IEEE Software, vol. 28, pp. 74-83, 2011.

[49] L. M. Rea and R. A. Parker, "Designing and conducting survey research: A comprehensive guide," John Wiley & Sons, 2014.

[50] J. Linåker, S. M. Sulaman, R. Maiani de Mello, and M. Höst, "Guidelines for conducting surveys in software engineering," 2015.

[51] H. M. Olague, L. H. Etzkorn, S. Gholston, and S. Quattlebaum, "Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes," IEEE Transactions on Software Engineering, vol. 33, 2007.

[52] R. K. Yin, "Case study research: Design and methods," Sage publications, 2013.

[53] P. Runeson, M. Host, A. Rainer, and B. Regnell, Case study research in software engineering: Guidelines and examples: John Wiley & Sons, 2012.

[54] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in Proceedings of the 28th International Conference on Software Engineering, 2006, pp. 452-461.

[55] N. Kiyavitskaya, N. Zeni, L. Mich, and D. M. Berry, "Requirements for tools for ambiguity identification and measurement in natural language requirements specifications," Requirements Engineering, vol. 13, pp. 207-239, 2008.

[56] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in Proceedings of the 30th International Conference on Software engineering, 2008, pp. 181-190.

[57] H. Femmer, D. M. Fernández, S. Wagner, and S. Eder, "Rapid quality assurance with requirements smells," Journal of Systems and Software, vol. 123, pp. 190-213, 2017.

[58] A. Van Lamsweerde, "Requirements engineering in the year 00: a research perspective," in Proceedings of the 22nd International Conference on Software Engineering, 2000, pp. 5-19.

[59] A. Mockus, N. Nagappan, and T. T. Dinh-Trong, "Test coverage and post-verification defects: A multiple case study," Empirical Software Engineering and Measurement, 2009. ESEM 2009. pp. 291-301.

[60] J. Itkonen and K. Rautiainen, "Exploratory testing: a multiple case study," Empirical Software Engineering, 2005.

[61] D. N. Card, "Learning from our mistakes with defect causal analysis," IEEE Software, vol. 15, pp. 56-63, 1998.

[62] V. S. Sheng, B. Gu, W. Fang, and J. Wu, "Cost-sensitive learning for defect escalation," Knowledge-Based Systems, vol. 66, pp. 146-155, 2014.

[63] M. I. Kamata and T. Tamai, "How does requirements quality relate to project success or failure?," Requirements Engineering Conference, 2007, pp. 69-78.

[64] W. A. Harrison and K. I. Magel, "A complexity measure based on nesting level," ACM Sigplan Notices, vol. 16, pp. 63-74, 1981.

[65] M. M. S. Sarwar, S. Shahzad, and I. Ahmad, "Cyclomatic complexity: The nesting problem," Digital Information Management (ICDIM), pp. 274-279.

[66] M. Shepperd, "A critique of cyclomatic complexity as a software metric," Software Engineering Journal, vol. 3, pp. 30-36, 1988.

[67] R. P. Buse and W. R. Weimer, "Learning a metric for code readability," IEEE Transactions on Software Engineering, vol. 36, pp. 546-558, 2010.

[68] N. E. Fenton and M. Neil, "Software metrics: roadmap," in Proceedings of the Conference on the Future of Software Engineering, 2000, pp. 357-370.

[69] B. Boehm, "A view of 20th and 21st century software engineering," in Proceedings of the 28th International Conference on Software engineering, 2006, pp. 12-29.
[70] T. Little, "Context-adaptive agility: managing complexity and uncertainty," IEEE Software, vol. 22, pp. 28-35, 2005.
[71] N. E. Fenton and S. L. Pfleeger, "Software metrics," Chapman & Hall London, 1991.
[72] N. E. Fenton and M. Neil, "A critique of software defect prediction models," IEEE Transactions on Software Engineering, vol. 25, pp. 675-689, 1999.
[73] C. Catal and B. Diri, "A systematic review of software fault prediction studies," Expert systems with applications, vol. 36, pp. 7346-7354, 2009.
[74] T. Fitz. (2009), "Continuous Deployment at IMVU: Doing the impossible fifty times a day," Available at: http://timothyfitz.wordpress.com/2009/02/10/continuous-deployment-at-imvu-doing-the-impossible-fifty-times-a-day/
[75] T. Chow and D.-B. Cao, "A survey study of critical success factors in agile software projects," Journal of Systems and Software, vol. 81, pp. 961-971, 2008.
[76] G. I. U. S. Perera and M. S. D. Fernando, "Enhanced agile software development - hybrid paradigm with LEAN practice," in International Conference on Industrial and Information Systems (ICIIS), 2007, pp. 239-244.
[77] D. Wisell, P. Stenvard, A. Hansebacke, and N. Keskitalo, "Considerations when Designing and Using Virtual Instruments as Building Blocks in Flexible Measurement System Solutions," in IEEE Instrumentation and Measurement Technology Conference, 2007, pp. 1-5.
[78] International Bureau of Weights and Measures, "International vocabulary of basic and general terms in metrology 2nd ed," Genève, Switzerland: International Organization for Standardization, 1993.
[79] J. Lawler and B. Kitchenham, "Measurement modeling technology," IEEE Software, vol. 20, pp. 68-75, 2003.
[80] F. Garcia, M. Serrano, J. Cruz-Lemus, F. Ruiz, M. Pattini, and ALARACOS Research Group, "Managing Software Process Measurement: A Meta-model Based Approach," Information Sciences, vol. 177, pp. 2570-2586, 2007.
[81] Harvard Business School, "Harvard business review on measuring corporate performance," Boston, MA: Harvard Business School Press, 1998.
[82] D. Parmenter, "Key performance indicators: developing, implementing, and using winning KPIs," John Wiley & Sons, 2007.
[83] R. L. Baskerville and A. T. Wood-Harper, "A Critical Perspective on Action Research as a Method for Information Systems Research," Journal of Information Technology, vol. 1996, pp. 235-246, 1996.
[84] G. I. Susman and R. D. Evered, "An Assessment of the Scientific Merits of Action Research," Administrative Science Quarterly, vol. 1978, pp. 582-603, 1978.
[85] P. Tomaszewski, P. Berander, and L.-O. Damm, "From Traditional to Streamline Development - Opportunities and Challenges," Software Process Improvement and Practice, vol. 2007, pp. 1-20, 2007.
[86] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang, "An industrial study on the risk of software changes," in Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, 2012, p. 62.
[87] M. Staron, W. Meding, G. Karlsson, and C. Nilsson, "Developing measurement systems: an industrial case study," Journal of Software Maintenance and Evolution: Research and Practice, vol. 23, pp. 89-107, 2011.

[88] M. Staron and W. Meding, "Ensuring reliability of information provided by measurement systems," Software Process and Product Measurement, Springer, 2009, pp. 1-16.
[89] W.-M. Han and S.-J. Huang, "An empirical analysis of risk components and performance on software projects," Journal of Systems and Software, vol. 80, pp. 42-50, 2007.
[90] B. Boehm, "Software risk management," Springer, 1989.
[91] L. Wallace, M. Keil, and A. Rai, "Understanding software project risk: a cluster analysis," Information & Management, vol. 42, pp. 115-125, 2004.
[92] S. Amland, "Risk-based testing: Risk analysis fundamentals and metrics for software testing including a financial application case study," Journal of Systems and Software, vol. 53, pp. 287-295, 9/15/ 2000.
[93] H. Barki, S. Rivard, and J. Talbot, "Toward an assessment of software development risk," Journal of management information systems, pp. 203-225, 1993.
[94] J. H. Iversen, L. Mathiassen, and P. A. Nielsen, "Managing risk in software process improvement: an action research approach," Mis Quarterly, vol. 28, pp. 395-433, 2004.
[95] R. M. Bell, T. J. Ostrand, and E. J. Weyuker, "Looking for bugs in all the right places," in Proceedings of the 2006 International Symposium on Software Testing and Analysis, 2006, pp. 61-72.
[96] G. Denaro, S. Morasca, M. Pezz, "Deriving models of software fault-proneness," presented at the Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, Ischia, Italy, 2002.
[97] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," IEEE Transactions on Software Engineering, , vol. 31, pp. 897-910, 2005.
[98] H. M. Olague, L. H. Etzkorn, S. Gholston, and S. Quattlebaum, "Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes," IEEE Transactions on Software Engineering, vol. 33, pp. 402-419, 2007.
[99] J. P. Hudepohl, S. J. Aud, T. M. Khoshgoftaar, E. B. Allen, and J. Mayrand, "Emerald: Software metrics and models on the desktop," IEEE Software, vol. 13, pp. 56-60, 1996.
[100] S. Kanmani, V. R. Uthariaraj, V. Sankaranarayanan, and P. Thambidurai, "Object-oriented software fault prediction using neural networks," Information and Software Technology, vol. 49, pp. 483-492, 2007.
[101] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," IEEE Transactions on Software Engineering, vol. 31, pp. 340-355, 2005.
[102] G. J. Pai and J. B. Dugan, "Empirical analysis of software fault content and fault proneness using Bayesian methods," IEEE Transactions on Software Engineering, vol. 33, pp. 675-686, 2007.
[103] M. Staron and W. Meding, "Monitoring Bottlenecks in Agile and Lean Software Development Projects – A Method and Its Industrial Use," in Product-Focused Software Process Improvement, Tore Cane, Italy, 2011, pp. 3-16.
[104] N. E. Fenton and S. L. Pfleeger, "Software metrics : a rigorous and practical approach," 2nd ed. London: International Thomson Computer Press, 1996.

[105] M. Staron, W. Meding, and C. Nilsson, "A framework for developing measurement systems and its industrial evaluation," Information and Software Technology, vol. 51, pp. 721-737, 2009.

[106] T. J. McCabe, "A complexity measure," IEEE Transactions on Software Engineering, pp. 308-320, 1976.

[107] G. Jay, J. E. Hale, R. K. Smith, D. Hale, N. A. Kraft, and C. Ward, "Cyclomatic complexity and lines of code: empirical evidence of a stable linear relationship," Journal of Software Engineering and Applications (JSEA), 2009.

[108] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," 30th International Conference on Software Engineering, ACM/IEEE, 2008, pp. 181-190.

[109] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto, "Software quality analysis by code clones in industrial legacy software," Eighth IEEE Symposium on Software Metrics, 2002, pp. 87-94.

[110] D. E. Neumann, "An enhanced neural network technique for software risk analysis," IEEE Transactions on Software Engineering, vol. 28, pp. 904-912, 2002.

[111] G. M. Selim, L. Barbour, W. Shang, B. Adams, A. E. Hassan, and Y. Zou, "Studying the impact of clones on software defects," 17th Working Conference on Reverse Engineering (WCRE), 2010, pp. 13-21.

[112] A. G. Koru, K. El Emam, D. Zhang, H. Liu, and D. Mathew, "Theory of relative defect proneness," Empirical Software Engineering, vol. 13, pp. 473-498, 2008.

[113] D. R. Cox, "Regression models and life-tables," Journal of the Royal Statistical Society. Series B (Methodological), pp. 187-220, 1972.

[114] I. Gondra, "Applying machine learning to software fault-proneness prediction," Journal of Systems and Software, vol. 81, pp. 186-195, 2// 2008.

[115] P. C. Pendharkar, "Exhaustive and heuristic search approaches for learning a software defect prediction model," Engineering Applications of Artificial Intelligence, vol. 23, pp. 34-40, 2// 2010.

[116] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai, "Comparing case-based reasoning classifiers for predicting high risk software components," Journal of Systems and Software, vol. 55, pp. 301-320, 1/15/ 2001.

[117] T. Bakota, P. Hegedus, P. Kortvelyesi, R. Ferenc, and T. Gyimóthy, "A probabilistic software quality model," 27th IEEE International Conference on Software Maintenance (ICSM), 2011, pp. 243-252.

[118] R. Baggen, J. P. Correia, K. Schill, and J. Visser, "Standardized code quality benchmarking for improving software maintainability," Software Quality Journal, vol. 20, pp. 287-307, 2012.

[119] V. Antinyan, M. Staron, W. Meding, A. Henriksson, J. Hansson, and A. Sandberg, "Defining Technical Risks in Software Development," International Conference on Software Process and Product Measurement (IWSM-MENSURA), 2014, pp. 66-71.

[120] E. Knauss and C. El Boustani, "Assessing the quality of software requirements specifications," 16th IEEE International Requirements Engineering Conference, 2008. pp. 341-342.

[121] D. Damian and J. Chisan, "An empirical study of the complex relationships between requirements engineering processes and other processes that lead to payoffs in productivity, quality, and risk management," IEEE Transactions on Software Engineering, vol. 32, pp. 433-453, 2006.

[122] M. Kauppinen, M. Vartiainen, J. Kontio, S. Kujala, and R. Sulonen, "Implementing requirements engineering processes throughout organizations: success factors and challenges," Information and Software Technology, vol. 46, pp. 937-953, 2004.

[123] S. Jacobs, "Introducing measurable quality requirements: a case study," IEEE International Symposium on Requirements Engineering, 1999, pp. 172-179.

[124] H. Mat Jani and A. Tariqul Islam, "A framework of software requirements quality analysis system using case-based reasoning and Neural Network," Information Science and Service Science and Data Mining (ISSDM), 2012 6th International Conference on New Trends in, 2012, pp. 152-157.

[125] C. Patel and M. Ramachandran, "Story card maturity model (SMM): a process improvement framework for agile requirements engineering practices," Journal of Software, vol. 4, pp. 422-435, 2009.

[126] S. Beecham, T. Hall, and A. Rainer, "Defining a requirements process improvement model," Software Quality Journal, vol. 13, pp. 247-279, 2005.

[127] F. Houdek, "Managing Large Scale Specification Projects," Keynote Presentation in Working Conference on Requirements Engineering: Foundation for Software Quality, https://refsq.org/2013/industry-track/presentations/, 2013.

[128] B. H. Cheng and J. M. Atlee, "Research directions in requirements engineering," Future of Software Engineering, 2007, pp. 285-303.

[129] B. Regnell, R. B. Svensson, and K. Wnuk, "Can we beat the complexity of very large-scale requirements engineering?," Requirements Engineering: Foundation for Software Quality, Springer, 2008, pp. 123-128.

[130] K. Wnuk, B. Regnell, and C. Schrewelius, "Architecting and coordinating thousands of requirements–an industrial case study," Requirements Engineering: Foundation for Software Quality, ed: Springer, 2009, pp. 118-123.

[131] N. P. Napier, L. Mathiassen, and R. D. Johnson, "Combining perceptions and prescriptions in requirements engineering process assessment: an industrial case study," IEEE Transactions on Software Engineering, vol. 35, pp. 593-606, 2009.

[132] ISO/IEC 9126 International Standard, 2000.

[133] E. Rechtin and M. W. Maier, "The art of systems architecting," CRC Press, 2000.

[134] Systems and software engineering Vocabulary, I. S. I. I. I. 24765, 2010.

[135] D. I. Sjøberg, B. Anda, and A. Mockus, "Questioning software maintenance metrics: a comparative case study," Proceedings of the ACM-IEEE International Symposium on Empirical Eoftware Engineering and Measurement, 2012, pp. 107-110.

[136] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," Journal of machine Learning research, vol. 3, pp. 993-1022, 2003.

[137] G. Génova, J. M. Fuentes, J. Llorens, O. Hurtado, and V. Moreno, "A framework to measure and improve the quality of textual requirements," Requirements Engineering, vol. 18, pp. 25-41, 2013.

[138] L. E. Hyatt and L. H. Rosenberg, "Software metrics program for risk assessment," Acta Astronautica, vol. 40, pp. 223-233, 1997.

[139] B. Gleich, O. Creighton, and L. Kof, "Ambiguity detection: Towards a tool explaining ambiguity sources," Requirements Engineering: Foundation for Software Quality, ed: Springer, 2010, pp. 218-232.

[140] V. Antinyan, M. Staron, W. Meding, P. Osterstrom, E. Wikstrom, J. Wranker, et al., "Identifying risky areas of software code in Agile/Lean software development: An industrial experience report," Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014, pp. 154-163.

[141] R. L. Baskerville, "Investigating information systems with action research," Communications of the AIS, vol. 2, p. 4, 1999.
[142] M. G. Kendall, "A new measure of rank correlation," Biometrika, vol. 30, pp. 81-93, 1938.
[143] R. L. Baskerville and A. T. Wood-Harper, "A critical perspective on action research as a method for information systems research," Journal of Information Technology, vol. 11, pp. 235-246, 1996.
[144] F. Fabbrini, M. Fusani, S. Gnesi, and G. Lami, "An automatic quality evaluation for natural language requirements," Requirements Engineering: Foundation for Software Quality REFSQ, 2001, pp. 4-5.
[145] J. Bøegh, "A New Standard for Quality Requirements," IEEE Software, vol. 25, pp. 57-63, 2008.
[146] A. Davis, S. Overmyer, K. Jordan, J. Caruso, F. Dandashi, A. Dinh, et al., "Identifying and measuring quality in a software requirements specification," Software Metrics Symposium, 1993. pp. 141-152.
[147] R. J. Costello and D.-B. Liu, "Metrics for requirements engineering," Journal of Systems and Software, vol. 29, pp. 39-63, 1995.
[148] R. Vlas and W. N. Robinson, "A rule-based natural language technique for requirements discovery and classification in open-source software development projects," 44th Hawaii International Conference on System Sciences (HICSS), 2011, pp. 1-10.
[149] C. Huertas and R. Juárez-Ramírez, "NLARE, a natural language processing tool for automatic requirements evaluation," International Information Technology Conference, 2012, pp. 371-378.
[150] J. Kasser, W. Scott, X.-L. Tran, and S. Nesterov, "A proposed research programme for determining a metric for a good requirement," Conference on Systems Engineering Research, 2006, p. 1.
[151] H. Femmer, D. M. Fernández, E. Juergens, M. Klose, I. Zimmer, and J. Zimmer, "Rapid requirements checks with requirements smells: two case studies," International Workshop on Rapid Continuous Software Engineering, 2014, pp. 10-19.
[152] E. Parra, C. Dimou, J. Llorens, V. Moreno, and A. Fraga, "A methodology for the classification of quality of requirements using machine learning techniques," Information and Software Technology, vol. 67, pp. 180-195, 2015.
[153] G. H. Subramanian, P. C. Pendharkar, and M. Wallace, "An empirical study of the effect of complexity, platform, and program type on software development effort of business applications," Empirical Software Engineering, vol. 11, pp. 541-553, 2006.
[154] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig, "Software complexity and maintenance costs," Communications of the ACM, vol. 36, pp. 81-95, 1993.
[155] A. Abran, "Software metrics and software metrology," John Wiley & Sons, 2010.
[156] E. J. Weyuker, "Evaluating software complexity measures," IEEE Transactions on Software Engineering, vol. 14, pp. 1357-1365, 1988.
[157] N. F. Schneidewind, "Methodology for validating software metrics," IEEE Transactions on Software Engineering, vol. 18, pp. 410-422, 1992.
[158] B. Kitchenham, S. L. Pfleeger, and N. Fenton, "Towards a framework for software measurement validation," IEEE Transactions on Software Engineering, vol. 21, pp. 929-944, 1995.
[159] S. Sarwar, M. Muhammd, S. Shahzad, and I. Ahmad, "Cyclomatic complexity: The nesting problem," Eighth International Conference on Digital Information Management (ICDIM), 2013, pp. 274-279.

[160] M. Shepperd and D. C. Ince, "A critique of three metrics," Journal of Systems and Software, vol. 26, pp. 197-210, 1994.

[161] J. Graylin, J. E. Hale, R. K. Smith, H. David, N. A. Kraft, and W. Charles, "Cyclomatic complexity and lines of code: empirical evidence of a stable linear relationship," Journal of Software Engineering and Applications, vol. 2, p. 137, 2009.

[162] C. Kaner, "Software engineering metrics: What do they measure and how do we know?," METRICS 2004. IEEE CS, 2004.

[163] J. C. Munson and T. M. Khoshgoftaar, "The detection of fault-prone programs," IEEE Transactions on Software Engineering, vol. 18, pp. 423-433, 1992.

[164] V. Antinyan, M. Staron, W. Meding, P. Österström, E. Wikstrom, J. Wranker, et al., "Identifying risky areas of software code in Agile/Lean software development: An industrial experience report," IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014, pp. 154-163.

[165] M. H. Halstead, "Elements of Software Science (Operating and programming systems series)," Elsevier Science Inc., 1977.

[166] S. Henry and D. Kafura, "Software structure metrics based on information flow," IEEE Transactions on Software Engineering, pp. 510-518, 1981.

[167] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," IEEE Transactions on Software Engineering, vol. 20, pp. 476-493, 1994.

[168] T. Tenny, "Program readability: Procedures versus comments," IEEE Transactions on Software Engineering, vol. 14, pp. 1271-1279, 1988.

[169] R. P. Buse and W. R. Weimer, "Learning a metric for code readability," IEEE Transactions on Software Engineering, vol. 36, pp. 546-558, 2010.

[170] P. Koopman, "A case study of Toyota unintended acceleration and software safety," Presentation. Sept, 2014.

[171] R. C. Seacord, D. Plakosh, and G. A. Lewis, "Modernizing legacy systems: software technologies, engineering processes, and business practices," Addison-Wesley Professional, 2003.

[172] S. Xiao, J. Witschey, and E. Murphy-Hill, "Social influences on secure development tool adoption: why security tools spread," in Proceedings of the 17th ACM conference on Computer supported cooperative work & social computing, 2014, pp. 1095-1106.

[173] P. Devanbu, T. Zimmermann, and C. Bird, "Belief & evidence in empirical software engineering," in Proceedings of the 38th International Conference on Software Engineering, 2016, pp. 108-119.

[174] D. P. Tegarden, S. D. Sheetz, and D. E. Monarchi, "A software complexity model of object-oriented systems," Decision Support Systems, vol. 13, pp. 241-262, 1995.

[175] R. R. Gonzalez, "A unified metric of software complexity: measuring productivity, quality, and value," Journal of Systems and Software, vol. 29, pp. 17-37, 1995.

[176] N. Chapin, "A measure of software complexity," Proceedings of the 1979 NCC, pp. 995-1002, 1979.

[177] J. C. Munson and T. M. Kohshgoftaar, "Measurement of data structure complexity," Journal of Systems and Software, vol. 20, pp. 217-225, 1993.

[178] H. Tao and Y. Chen, "Complexity measure based on program slicing and its validation," Wuhan University Journal of Natural Sciences, vol. 19, pp. 512-518, 2014.

[179] Y. Yang, Y. Zhou, H. Lu, L. Chen, Z. Chen, B. Xu, et al., "Are slice-based cohesion metrics actually useful in effort-aware post-release fault-proneness prediction?

An empirical study," IEEE Transactions on Software Engineering, vol. 41, pp. 331-357, 2015.

[180] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "DECOR: A method for the specification and detection of code and design smells," IEEE Transactions on Software Engineering, vol. 36, pp. 20-36, 2010.

[181] S. Kpodjedo, F. Ricca, P. Galinier, Y.-G. Guéhéneuc, and G. Antoniol, "Design evolution metrics for defect prediction in object oriented systems," Empirical Software Engineering, vol. 16, pp. 141-175, 2011.

[182] Y. Wang and J. Shao, "Measurement of the cognitive functional complexity of software," The Second IEEE International Conference on Cognitive Informatics, 2003, pp. 67-74.

[183] S. N. Waweru, W. Mwangi, and W. Joseph, "A Software Code Complexity Framework; Based on an Empirical Analysis of Software Cognitive Complexity Metrics using an Improved Merged Weighted Complexity Measure," International Journal of Advanced Research in Computer Science, vol. 4, 2013.

[184] G. Keshavarz, N. Modiri, and M. Pedram, "Metric for Early Measurement of Software Complexity," Interfaces, vol. 5, p. 15, 2011.

[185] M. Al-Hajjaji, I. Alsmadi, and S. Samarah, "Evaluating software complexity based on decision coverage," LAP LAMBERT Academic Publishing, 2013.

[186] S. D. Suh and I. Neamtiu, "Studying software evolution for taming software complexity," 21st Australian Software Engineering Conference (ASWEC) 2010, pp. 3-12.

[187] N. Salman, "Complexity metrics as predictors of maintainability and integrability of software components," Cankaya University Journal of Arts and Sciences, vol. 1, 2006.

[188] Y. Kanellopoulos, P. Antonellis, D. Antoniou, C. Makris, E. Theodoridis, C. Tjortjis, et al., "Code quality evaluation methodology using the ISO/IEC 9126 standard," preprint arXiv:1007.5117, 2010.

[189] V. Antinyan, M. Staron, J. Hansson, W. Meding, P. Österström, and A. Henriksson, "Monitoring Evolution of Code Complexity and Magnitude of Changes," Acta Cybernetica, vol. 21, pp. 367-382, 2014.

[190] G. J. Myers, C. Sandler, and T. Badgett, "The art of software testing," John Wiley & Sons, 2011.

[191] A. Glover, "In pursuit of code quality: Don't be fooled by the coverage report," IBM Developer Works blog post, pp. 1-2, 2006.

[192] B. Marick, "How to misuse code coverage," in Proceedings of the 16th Interational Conference on Testing Computer Software, 1999, pp. 16-18.

[193] Y. Chernak, "Validating and improving test-case effectiveness," IEEE software, vol. 18, pp. 81-86, 2001.

[194] J. Voas, "How assertions can increase test effectiveness," IEEE Software, vol. 14, pp. 118-119, 1997.

[195] M. R. Lyu, J. Horgan, and S. London, "A coverage analysis tool for the effectiveness of software testing," IEEE Transactions on Reliability, vol. 43, pp. 527-535, 1994.

[196] B. Smith and L. A. Williams, "A survey on code coverage as a stopping criterion for unit testing," North Carolina State University. Dept. of Computer Science, 2008.

[197] L. Briand and D. Pfahl, "Using simulation for assessing the real impact of test coverage on defect coverage," 10th International Symposium on Software Reliability Engineering, 1999, pp. 148-157.

[198] P. S. Kochhar, F. Thung, and D. Lo, "Code coverage and test suite effectiveness: Empirical study with real bugs in large systems," IEEE 22nd International Conference on in Software Analysis, Evolution and Reengineering (SANER), 2015, pp. 560-564.

[199] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," 36th International Conference on Software Engineering, 2014, pp. 435-445.

[200] X. Cai and M. R. Lyu, "The effect of code coverage on fault detection under different testing profiles," ACM SIGSOFT Software Engineering Notes, vol. 30, pp. 1-7, 2005.

[201] G. Gay, M. Staats, M. Whalen, and M. P. Heimdahl, "The risks of coverage-directed test case generation," IEEE Transactions on Software Engineering, vol. 41, pp. 803-819, 2015.

[202] A. Sellami and A. Abran, "The contribution of metrology concepts to understanding and clarifying a proposed framework for software measurement validation," 13th International Workshop on Software Measurement (IWSM), Montreal, Canada, 2003, pp. 18-40.

[203] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," Journal of systems and software, vol. 23, pp. 111-122, 1993.

[204] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," IEEE Transactions on Software Engineering, vol. 22, pp. 751-761, 1996.

[205] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," IEEE Transactions on Software Engineering, vol. 38, pp. 1276-1304, 2012.

[206] L. Briand, K. El Emam, and S. Morasca, "On the application of measurement theory in software engineering," Empirical Software Engineering, vol. 1, pp. 61-88, 1996.

[207] C. Mair and M. Shepperd, "Human judgement and software metrics: vision for the future," 2nd international workshop on emerging trends in software metrics, 2011, pp. 81-84.

[208] A. Meneely, B. Smith, and L. Williams, "Validating software metrics: A spectrum of philosophies," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 21, p. 24, 2012.

[209] J. McGarry, "Practical software measurement: objective information for decision makers," Addison-Wesley Professional, 2002.

[210] S. G. Elbaum and J. C. Munson, "Getting a handle on the fault injection process: validation of measurement tools," Fifth International Software Metrics Symposium, 1998, pp. 133-141.

[211] ISO/IEC 15939 international standard of software measurement, 2001.

[212] ISO/IEC 25020 international standard of software and system engineering, 2007.

[213] K. El-Emam, "A methodology for validating software product metrics," Encyclopaedia of software engineering, 2002.

[214] D. I. Sjøberg, J. E. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanovic, N.-K. Liborg, et al., "A survey of controlled experiments in software engineering," IEEE Transactions on Software Engineering, vol. 31, pp. 733-753, 2005.

[215] B. Boehm, R. Valerdi, J. Lane, and A. Brown, "COCOMO suite methodology and evolution," CrossTalk, vol. 18, pp. 20-25, 2005.

[216] A. Monden, D. Nakae, T. Kamiya, S.-i. Sato, and K.-i. Matsumoto, "Software quality analysis by code clones in industrial legacy software," Eighth IEEE Symposium on Software Metrics, 2002, pp. 87-94.

[217] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process," 7th joint meeting of the european software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, 2009, pp. 91-100.

[218] H. Zhang, X. Zhang, and M. Gu, "Predicting defective software components from code complexity measures," 13th Pacific Rim International Symposium on Dependable Computing, 2007, pp. 93-96.

[219] R. Reicherdt and S. Glesner, "Slicing MATLAB simulink models," 34th International Conference on Software Engineering (ICSE), 2012, pp. 551-561.

[220] P. Boström, R. Grönblom, T. Huotari, and J. Wiik, "An approach to contract-based verification of Simulink models," ed: Tech. Rep. 985, TUCS, 2010.

[221] J. Prabhu, "Complexity Analysis of Simulink Models to improve the Quality of Outsourcing in an Automotive Company," Manuscript, August, 2010.

[222] M. Olszewska, "Simulink-specific design quality metrics," Technical report, Turku, Finland, 2011.

[223] J. H. Iversen, L. Mathiassen, and P. A. Nielsen, "Managing risk in software process improvement: an action research approach," Mis Quarterly, pp. 395-433, 2004.

[224] D. L. Moody, "Metrics for evaluating the quality of entity relationship models," Conceptual Modeling–ER'98, ed: Springer, 1998, pp. 211-225.