Empowering Technical Risk Assessment in Software Development

Vard Antinyan

Department of Computer Science and Engineering

UNIVERSITY OF GOTHENBURG

Gothenburg 2014
Thesis for the degree of licentiate of engineering
Empowering Technical Risk Assessment in Software Development
© Vard Antinyan 2014
vard.antinyan@gu.se
Printed in Gothenburg, Sweden 2014
“Risk comes from not knowing what you’re doing.”
Warren Buffett
Abstract

Contemporary software development organizations focus on continuous delivery of product features, in order to respond customers’ requests quickly. In large software development organizations this focus is realized by parallel development of multiple features by independent software development teams. However, the continuous delivery is not only a contributory factor for organizational success but also a challenging one, in terms of delivering high quality product: Continuous code delivery combined with large code base induces risks of gradually degrading code quality. In order to quickly assess these risks software engineers need methods and automated tool support.

In this thesis we investigate the possibilities of assessing the technical risks, that the newly delivered code can be defect-prone or hard-to-maintain. We conduct a series of action research projects in two organizations at Ericsson and Volvo Group Truck Technology. Our results show that a set of complexity and change measures of source code can be used to assess the technical risks. Based on these measures, we develop methods and tools for technical risk assessment in source code delivery. These methods and tools empower software engineers to quickly identify the few risky functions and files in a large code base. Subsequently, software engineers can focus on enhancing product quality by refactoring these functions and files.

Keywords: software development, technical risk, risk assessment, software metrics, software complexity, continuous integration
Acknowledgement

I express my deep appreciation to my main advisor, Miroslaw Staron, who has guided me throughout my entire research. I also show my gratitude to my second advisor, Anna Sandberg, for her support in my research. I thank Jörgen Hansson for his support in this research journey. Miroslaw, Anna, and Jörgen, I am thankful to you all for your support in accomplishing this research project.

I cordially thank Wilhelm Meding for welcoming me to his team as a researcher at Ericsson. When I walked in through the doors of Ericsson, I knew nothing about large software organizations. It is thanks to Wilhelm’s dedication that I learned rapidly. I also thank Anders Henriksson for including me to his team as a researcher at Volvo Groups. Supporting with tools within counted hours and pointing to the right people to talk to are the greatest thing a researcher can get in an organization. I thank Per Österström. He is a person, being a project leader, with vast interest in research and collaboration. Per has invested so much from his precious time to take our research forward. Also, I would like to extend my thanks to Kent Niesel and Christoffer Högland for their participation and advices in this research. I am happy we are going to have more active collaboration with you in upcoming research projects. I thank Micael Caiman who supported me in getting access to Ericsson and do research inside the company. That has been a great privilege for me.

Other colleagues in industry: Jesper Derehag, Erik Wickström, Henrik Bergenwal, John Walmark, Johan Wranker, Mattias Runsten, Jonas Bjurhede and Jimmie Arvidsson. You helped me tremendously with any technical question that I had. Without your support, I hardly imagine my research targets would have been accomplished.

I thank colleagues in Software Engineering divisions and particularly Rebecca Cyren for creating nice environment for my research.

The most important people in my life: Ulrika - thank you for your unconditional support in bad and good times. Parents, Misha and Roza - you were just great support. Harutyun - thanks for genuinely sharing my problems and doubling my joys throughout this journey. Hasmik, Vahan, Artyom, Erik, Hrach, Davit, Edward and Hakob - you all were just great being beside me.
Included Publications

This thesis is based on the following studies.


IV. Vard Antinyan, Miroslaw Staron, Jesper Derehag, Mattias Runsten, Wilhelm Meding, “A method for effective reduction of code complexity by investigating various aspects of code complexity”. In submission to 22nd IEEE international conference on software analysis, evolution and reengineering

Additional Publications

Table of Contents

Abstract..............................................................................................................................................v
Acknowledgement ............................................................................................................................vii

Part 1: Technical Risks in Large Software Product Development .................................................................15
1 Introduction .......................................................................................................................................16
2 Research Overview ...........................................................................................................................18
3 Agile software development .............................................................................................................19
  3.1 Agile software development in the collaborating organizations .............................................19
4 Industrial Partners ..........................................................................................................................21
  4.1 Volvo Group Truck Technology (GTT) ......................................................................................21
  4.2 Ericsson ......................................................................................................................................21
  4.3 Volvo Cars Corporation (VCC) ..................................................................................................22
  4.4 Saab Defense ..............................................................................................................................23
5 Risk ..................................................................................................................................................23
  5.1 Risk assessment in software development ..............................................................................24
  5.2 Technical risks in software development .................................................................................26
    5.2.1 Technical risks in source code delivery .............................................................................27
6 Software metrics .............................................................................................................................30
  6.1 The scope of software metrics ..................................................................................................31
7 Research focus ..................................................................................................................................32
8 Research methodology ....................................................................................................................34
  8.1 Action research methodology ....................................................................................................34
  8.2 Application of action research .................................................................................................36
  8.3 Methods of data collections .......................................................................................................37
  8.4 Methods of data Analysis ..........................................................................................................38
    8.4.1 Correlation Analysis ............................................................................................................39
    8.4.2 Correlograms .......................................................................................................................40
    8.4.3 Time series ..........................................................................................................................40
    8.4.4 Reference Groups .................................................................................................................40
9 Validity Evaluation Frameworks used in the thesis .........................................................................41

Part 2: Defining Technical Risks in Software Development ............................................................................43
1 Introduction .......................................................................................................................................45
2 Motivation of the study ...................................................................................................................46
3 Defining Technical Risks ................................................................................................................47
4 Software Engineers View on Technical Risks ..............................................................................50
5 Risk Assessment by Forecast Model .............................................................................................53
6 Related Work ....................................................................................................................................55
7 Conclusions .......................................................................................................................................56

Part 3: Monitoring Evolution of Code Complexity and Magnitude of Changes ..................................................57
1 Introduction .......................................................................................................................................59
2 Related Work .....................................................................................................................................60
3 Design of the Study ..........................................................................................................................61
  3.1 Studied Organizations .................................................................................................................61
  3.2 Units of Analysis ..........................................................................................................................62
  3.3 Reference Group ..........................................................................................................................62

xi
Acknowledgment .......................................................................................................................... 118

Part 6: Profiling Pre-Release Software Product Performance .......................... 119

1  Introduction .............................................................................................................................. 121
2  Profiling .................................................................................................................................. 121
3  Profiling pre-release product and process performance in large software development organizations .......................................................... 123
4  Establishing the process of profiling ..................................................................................... 124
5  Case study: developing risk profile of software code .......................................................... 126
6  Case study: profiling change frequency of models over time ........................................ 131
7  Related work ............................................................................................................................ 133
8  Conclusions .............................................................................................................................. 135
References .................................................................................................................................... 137
PART 1: TECHNICAL RISKS IN LARGE SOFTWARE PRODUCT DEVELOPMENT
1 INTRODUCTION

A central issue in large software development projects is the assessment of quality of software development products. Based on such assessments developers can get insights on products’ time-to-market and customers’ satisfaction. Moreover, based on such assessments, products’ ineffectual design solutions can be identified and improved. Nowadays, large software development products contain several millions lines of code developed by tens of software engineering teams incorporated in one big organization. Large software code bases and sophisticated development processes require rigorous assessment methods and supporting tools adapted to the organization. Without assessments, the organization takes on risks of gradual degrading the quality of the software products and processes. However, as software products grow in size to provide more functionality for its customers, this kind of assessment certainly becomes an effort-intensive task if done manually.

The outcome of accepting such risks can be production delay, post-release product defects, and ultimately customers’ dissatisfaction. To balance the trade-offs between the costs of assessments and the prevention of gradual degradation of the product quality the companies usually conduct risk management. De Bakker [1] shows how risk management affects project success positively by permitting instrumental actions. Many researchers’ endeavors were directed towards creating a set of such instruments for effective risk management: Boehm [2], Charette [3], and Higuera and Haimes [4] have become standard for software engineers. Notwithstanding, with growing software product size the risk assessment activities need to be done more effectively. This requires automated support and increased focus on technical solutions used in the product. Assessing the technical solutions of the product, which was not considered so crucial for software production 15-20 years ago, have become an indispensable activity today [5]. This is due to the need of creating highly modular systems in order to manage risks due to the growing size of the products [5, 6]. The emerging risks due to poor design and engineering is defined as technical risks [7]. Poor technical solutions trigger technical risks of emerging defects in software code and increasing software maintenance effort. In order to manage these technical risks, software engineers in industry attempt to assess the quality of product’s design solutions, but the lack of standardization
Technical risks in large software product development

often impedes these attempts. Realizing technical risk management is a difficult task, and reasons for this difficulty are multifold:

- Inaccurately perceived nature of technical risks
- Incomprehensive foundation of its assessment and mitigation methods
- Incomplete automation of risk identification and assessment activities
- Unpredictable evolution of technical solutions

Fenton and Neil [8] state that one effective way of assessing risks is to use software metrics in order to provide quantitative support for risk mitigation. The use of software metrics permits automation of risk assessment, whereas, software metrics are mostly used for isolated purposes, such as defect prediction or cost estimation.

The research presented in this thesis proposes metrics-based practical methods for supporting technical risk assessment and automation in large software product development. The focus of this work is the identification and assessment of technical risks with source code delivery. The general research question that is addressed in the thesis is formulated as follows:

*How to automatically assess technical risks with source code delivery?*

In this thesis, first, a number of examples of technical risks are presented, which software engineers encounter in large software development products. Next, a scrutinized definition of software development technical risk is formulated. Last, a deeper investigation of technical risks in source code delivery is presented:

- Several code properties are examined that affect technical risks
- A set of metrics are selected for measuring these properties
- Metrics-based methods are proposed for risk assessment

Furthermore, tools for automation of risk assessment are also presented, that were developed in the collaborating organizations. In the end, we discuss the benefits for the collaborating organizations.

In the next section, a brief overview of the research topic is presented to help the reader understand the research context from an abstract level.
2 RESEARCH OVERVIEW

The aim of this research is to provide methods and tools for assessing technical risks with source code delivery in large software development products. Figure 1 illustrates the overview of our research in the context of software engineering. On the left side of the figure we can see the granular representation software development activities. The code delivery is the activity on which this research has focused. On the bottom of the figure we can see the conception of risks that software engineers deal with in different activities. The focus of this research is technical risks in code delivery, as represented on the lowest hierarchical order in the figure.

![Figure 1 An overview of the research context](image)

On the right hand of the figure we can see software metrics. This represents measures, by means of which various aspects of development products and processes are assessed. The source code metrics are used in this research for investigating the possibilities of assessing technical risks in source code delivery.

In the middle part of the figure the research focus is depicted in a rectangle. As the figure illustrates source code metrics are used in code delivery for assessing technical risks.

In Figure 1 the names of outlined areas also contain the section numbers (for example S.6 – section six) where the specified area is presented. The rest of part 1 is organized as follows: In section three, we present an overview of Agile Software Development in large software
product development. In section four we introduce the collaborating organizations. In section five and six, we introduce software risks and metrics correspondingly. In section seven, we present the research focus and break down the general research question into specific areas. Finally, in section eight we present the action research methodology by which we carried out this research.

3 AGILE SOFTWARE DEVELOPMENT

In recent years many software development organizations have adopted Agile software development methodology, which promises to accelerate software delivery to its customers [9, 10]. The organizations that we collaborate with has also adopted this methodology, therefore an overview of it is presented here.

The software development processes should not be built upon rigid methods, but rather flexible and change-driven principles. Agile manifesto suggests twelve such principles for software development [11]. The first principle states that:

*The customer satisfaction is the highest priority which can be achieved by early and continuous delivery*

Our collaborating companies had a strong focus on continuous delivery. With continuous delivery the quality of the software product also should be monitored and improved continuously, therefore the eight principle of agile manifesto states that:

*Continuous attention to technical excellence and good design enhances agility*

In continuous source code delivery the technical risk assessment should be carried out also continuously, in order to ensure that the code quality does not decrease gradually. In the next section we present how the collaborating organizations follow Agile principles.

3.1 Agile software development in the collaborating organizations

In our collaborating organizations, the size of the software products reach to several millions of lines of code and up to several hundred software engineers distributed in standalone development teams. A development team contains five to eight software engineers with a designated team leader within the team. Every development team have an assigned development area, such as development of a newly specified
feature or error correction in a previously developed area. To a certain degree, the development teams are given the mandate to make decisions and develop their own schedule for their development activities. In this process the project managers coordinate the activities for a common vision and ensure the amicable communication between the development teams. This type of behaviors is very much in line with original Agile development methodology.

An overview of software development process in collaborating organizations is illustrated in Figure 2. As the figure shows customers are continuously requesting new product features. These requests are formalized in a requirements specification (Req.), which was broken down into work packages (WP). The WPs are distributed in Agile software development teams, who design, test, integrate, and deliver the software.

![Figure 2 Software development and continuous delivery](image)

The continuous delivery process is supported by parallel development carried out by independent teams. This is reflected by arrows in the Figure 2, indicating parallel delivery by development teams. The process of customers’ request and feature delivery is a smooth continuous process aligned with the principles of Agile software development. With every new feature request customers also report feedback on the quality of previously delivered software. This means that identified post-release defects are also continuously reported. Accordingly, the organizations have assigned teams for continuous defect correction and quality improvement. Therefore, for the quality improvement teams the technical risk mitigation is crucial, in order to decrease the chance of getting high number of post-release defects. Hence, it is important to have effective tools for assessing the code quality and particularly defect-proneness before code delivery. This permits software engineers to identify the most defect-prone areas of
source code and improve it, so the risk of high number of post-release defects can be low.

4 INDUSTRIAL PARTNERS

In this section we give a brief overview of the collaborating organizations that support and are the direct stakeholders of the conducted research. Two of the organizations, Ericsson and Volvo Group Truck Technology, are the main collaborating partners and supporters of this research. The other two organizations, Volvo Cars Corporation and Saab, are partially involved in this research.

4.1 Volvo Group Truck Technology (GTT)

At Volvo GTT the software development organization supporting this research develop Electronic Control Units (ECU) for trucks. Every ECU represents a small computer, which performs various real-time tasks for the truck. Such tasks can be speed control, automatic breaking, air suspension, and climate control. Our collaborating organization develops one of the main ECUs for the truck. Additionally they carry out the verification and validation task of another ECU development (developed by another unit of the organization).

The development organization comprises tens of software engineers who perform requirements documentation, design, code development and testing. Each of these activities are executed by dedicated software development teams consisting of 5-8 software engineers.

The collaborating engineers of the organization have supported us with problem formulation, creating researcher-engineer collaboration environment in the organization and providing access to their data repositories. Additionally, four of the software engineers have been directly involved in evaluating the results. Several other engineers have been indirectly involved in the research by providing knowledge for tool usage and data collection.

4.2 Ericsson

The software development organization at Ericsson develops software for a telecom node. The node serves for interconnecting mobile users. The software of the node encapsulates a few millions lines of code. The functionality perform tasks such as signal processing, converting
signals from one type to another, controlling access of different user groups, and accounting.

The software development organization of this product comprises a few hundred software engineers. Tens of developers, distributed in semi-autonomous development teams, develop and test the software. For different architectural areas of the product, there are dedicated design architects. Development teams can carry out different activities such as maintenance, error correction or feature development. The assignments of the teams can change over time considering overall development efficiency.

We have conducted our research in the metrics team of the organization. The metrics team provides various measurement systems for measuring several aspects of the product’s performance. The metrics team leader has helped us with accessing data repositories. The line manager and five software engineers of the product have directly been involved and supported our research. The line manager has also been the main stakeholder for our action research project. Many other engineers that have not been directly involved in the research have helped us with accessing the right data repositories and the right versions of the product.

### 4.3 Volvo Cars Corporation (VCC)

VCC developed and manufactured personal cars. This development included software development of ECUs for cars also. Our collaboration organization develops one specific ECU for the car.

The development organization in VCC consists of tens of software engineers in several development teams. Every team is responsible for a particular activity of development. The activities are requirement specification, product design by domain specific language and testing.

The technical leader has been involved in the research. He has also provided access to company’s premises and data repositories. VCC has participated in the research presented in Part two and Part six of this thesis. In Part two, the technical leader from VCC have had a contribution to identifying technical risks. In Part six, we describe how one of the metrics developed at Ericsson is used at VCC.
4.4 Saab Defense

Saab Defense develops defense equipment. Our collaboration organization develops surveillance radar system.

The organization comprises several dedicated development teams, who carry out requirements specification, code design testing and deployment of the software.

Two software engineers have been involved in our research. They have supported the research presented in Part two of this thesis. They have also used the research results presented in Part four. As the organization develops defense equipment, there are strict rules for data collection and analysis. Therefore, we provided the tools to collaborating engineers who did data analysis themselves.

5 RISK

Historically the concept of risk is used to describe situations in decision making, where the possibility of loss exists. According to Slovic [12] the human perception of risk is closely associated with the factor of “dread”, threatening to inflict loss of health, property or reputation. Stanford Encyclopedia of Philosophy presents the most widely used five definitions of risk [13]:

1. An unwanted event which may or may not occur
2. The cause of an unwanted event which may or may not occur
3. The probability of an unwanted event which may or may not occur
4. The statistical expectation value of an unwanted event which may or may not occur
5. The fact that a decision is made under conditions of known probabilities

In ISO 31000 a risk is defined as an effect of uncertainty on objectives [14]. As we can see all definitions contain two elements: The first one is the uncertainty that an unwanted event can happen. The second one is the effect, impact, or cause of the unwanted event, which inflicts loss, when attempting to meet some targeted objectives. Kaplan and Garrick [15] define risk as a superposition of uncertainty and loss when taking on a decision. Unlike the previous definitions Kaplan and Garrick do not regard risk as a variable which should have a fixed value. They consider risk as collection of probabilities and losses, any instance of which can
occur. Figure 3 illustrates the risk according to the definition of Kaplan and Garrick: The risk is not a specific value of probability and loss but the whole surface of distribution of probabilities and losses.

![Figure 3 Risk as a superposition of probability and loss](image)

In this study we rely on Kaplan’s and Garrick’s definition of risk, because it assumes that before the occurrence of the unwanted event the loss cannot be a fixed value. The loss in source code delivery is the number of defects and decreased maintainability of the code. Therefore, when assessing technical risks of source code delivery, the loss is not a fixed value before the delivery is done. We do not usually say source code will or will not have defects, or the code is maintainable or not maintainable. We rather talk about the number of defects and degree of maintainability of delivered code.

The last phrase of risk definition in Kaplan’s and Garrick’s definition focuses on decision making. In business organizations the concept of risk is usually accompanied with decision making, because when making decision, the organization is prompted to change from one state to another. Notwithstanding, the decision maker cannot forecast the exact consequences of the decision, thus leaving other unpredictable possibilities. Not making any decision is another decision, therefore there is no such thing as avoiding risks but rather choosing between risks.

### 5.1 Risk assessment in software development

In software development organizations risk is viewed as a possibility of loss when making decisions under given uncertainty [15]. Several researchers have investigated the most common risks in software engineering and their nature [2, 16]. Risks identified in their work are divided into categories as reputational, operational, business, technological, and technical. Software risk management is usually carried out in software projects by the project manager or risk manager
with a participation of other software engineers and experts. Phases of software risk management include:

1. Identification
2. Prioritization
3. Assessment
4. Mitigation strategy planning
5. Implementation of a strategy plan

In this thesis we focus on the assessment phase which usually is carried out manually with software engineers and experts. Normally in an organized workshop, the experts (e.g. architects, managers, business analysts) identify and present risky events (phase 1). They then select top priority risks (phase 2) and assess them (phase 3) [4]. In general there are several assessment methods of risks, both quantitative and qualitative, two of which are mostly used. The first one relies on qualitative assessment of risk, where the likelihood and the impact of the identified adverse event are qualitatively assessed. Then these two values are summed or multiplied as a relative value of risk exposure, [17] (Figure 4).

\[ \text{Risk} = \text{Likelihood} \times \text{Loss} \quad (1) \]

<table>
<thead>
<tr>
<th>Likelihood</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mark</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>12</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>25</td>
</tr>
</tbody>
</table>

**Figure 4 Qualitative assessment of risk**

The other assessment method relies on the triple estimation formula, and is fundamentally a different way of risk estimation compared to the first one. It allows not only estimating the absolute value of loss, but also the most likely interval of loss [18]. According to this method the value of risk exposure is calculated as:

\[ \text{Risk} = \frac{(\text{max} + 4 \times \text{most likely} + \text{min})}{6} \quad (2) \]

Where max is the maximum estimated loss, most likely is the most likely estimated loss, and min is the minimum estimated loss by experts. In this case, not only the value of risk exposure matters, but also the [min,
max] interval. This method has been and still is used at Ericsson [19]. After the assessment mitigation strategy plan is prepared (phase 4) and executed by the initiation of the risk facilitator or project manager (phase 5).

In the next section we introduce technical risks in software engineering, discuss the difficulties of their assessment with aforementioned techniques, and emphasize the need of new approaches for technical risk assessment.

## 5.2 Technical risks in software development

A technical risk is defined similar to other definitions of risks presented in section 5. The risk is a combination of likelihood of adverse event and expected loss. Educational websites define technical risks as follows:

1. Investor Words defines technical risks as *the probability of loss incurred through the execution of technical process in which the outcome is uncertain* [20]. Technical risk assessment in this case, is realized based on prior experiences.
2. Business dictionary defines technical risk as *an exposure to loss due to design and engineering of a product* [7].

In software development, the technical risk is defined as *a measure of probability and severity of adverse effects that are inherent in the development of software, and associated with its intended functions and performance requirements* [21]. All of these definitions of technical risks rely on the generic risk definitions, implying that there are two orthogonal components in risk, probability and loss. In case of technical processes, the probability and loss are induced specifically by technical decisions. In order to make a good decision the engineers need an objective and quantitative assessment of risk, permitting clearly determined choice on how to spend the precious time for enhancing development productivity. The risk quantification formulas presented in section 5.1 are not applicable for certain technical risks because of three essential reasons:

1. The number of risk items to be assessed are overwhelmingly bigger than in other types of risks
2. The frequency of risk assessment is needed on a daily or even sometimes hourly basis
3. Before the technical risk occurs we cannot give an exact value of *loss*, but an interval: After the risk occurs we can calculate the actual value of the loss.

The first two reasons highlight the importance of automation of risk assessment. Thus we need a method which supports the automatic risk assessment. The last reason indicates that we need a method which does not imply preliminarily decided value of expected loss.

**5.2.1 Technical risks in source code delivery**

Technical risk assessment in source code delivery is the focus of this research. We therefore give an overview of technical risks in the delivery process.

In the collaborating organizations, the development teams continuously deliver newly developed software code [22]. They merge the newly developed code with the whole product code that is called an integration branch. The latest updates of the source code are always merged with an integration branch called the main integration branch. Beside this, due to several possible releases and activities on the product, there are several other integration branches, such as release branch for a particular platform, error correction branch, and testing branch. Figure 5 presents several possible integration branches in code delivery.

![Figure 5: Risks in code development branches](image)

The black circles show the place where a new branch was created for a specific activity. The stripped circles show the place where the newly developed or modified code was merged back to its initial branch.
In the merging circles the dashed arrows show the risks which the development team takes when merging the newly developed code. One of these risks can be the likelihood of new defects inserted in the code together with the merged code.

However, the newly developed code can also become more risky for multiple reasons. For example the increased code complexity or too many changes in the code can trigger risks. Figure 6 presents an example of triggers of technical risks and the expected outcome, when merging a newly developed portion of code.

![Figure 6 Triggers of technical risk and outcomes](image)

In the process of source code development and delivery to integration branches there are there major technical risks that developers are concerned with.

1. Does the newly delivered code contain defects?
   a. How many defects does it contain?
   b. How severe are the defects?
2. Does the newly delivered code contain an area that is difficult to maintain or has become significantly less maintainable than it was before?
3. Is the newly delivered code at least as manageable as it was before?

As we can conclude from the listed questions, three main adverse events are associated with the integration of newly developed or maintained code, the occurrence of which eventually will slow down the development. The possibility of such adverse events, as inserted errors or exacerbated maintainability of source code, in standard terms can be stated as technical risks associated with source code delivery.
In Agile software development, where the product features are delivered to the customer continuously, the number of newly developed code portions and the number of merges of new code to the main integration branch per day can reach to hundreds. In order to assess technical risks in source code delivery, there is a need of methods that permit automation of risk assessment.

In software product development one choice in technical decision making can be integrating newly developed code with the whole product code. An alternative choice can be doing more testing and integrating the code later. In the above example there is a trade-off with the aforementioned two alternative choices: Integrating the code early with a risk of newly inserted errors versus integrating late with a risk of late integration errors (Figure 7).

![Figure 7 Illustrating trade-offs between two technical choices in decision making](image)

Figure 7 shows that when code integration is late, changes to the overall code are introduced by other teams (dashed arrows). Thus there is a risk that integration will not be smooth due to changes that other teams introduced to the overall code. Pondering on this decision, software engineers need to understand how complex the newly developed code is, how much problems it might cause, how much time and effort it will take to reduce the complexity, and how the late integration will affect the business of the organization. Based on this considerations software engineers attempt to understand the trade-off between the two choices and find an optimal point when the code can be integrated to the main branch.
6 SOFTWARE METRICS

One effective way of obtaining information about various aspects of software is observing its attributes (properties). *Software attributes (properties) are characteristics of an entity that can be distinguished quantitatively or qualitatively by human or automated means* [23].

For example, size, complexity or change rate are properties of software code. Length, size, documenting style and understandability are properties of textual requirements. The experience shows that measurable properties of software are widely used for observing the behavior of the software products. Measures that are designed to quantify properties of software artifacts are also called software metrics. For example, Lines of Code is a metric for measuring software code’s size. The ISO/IEC 15939 Standard for Software Quality Metrics defines the measurement as the process of assigning a number or category to an entity to describe an attribute (property) of that entity. A metric is a variable to which a value is assigned as a result of measurement [23]. An example entity can be the source code of a product with such attributes (property) as code complexity, size, length, readability, etc. A measure of the size can be the calculated value of number of lines of code of a file. Figure 8 illustrates how software properties defined in the empirical world can be transmitted into quantified measures in the relational world.

![Figure 8 Mapping software properties from empirical world to relational world](image-url)
Software artifacts, source code files, have several properties: Complexity, size, change frequency, etc. In order for this properties to be quantifiable and comparable, software metrics are designed.

The lower part of Figure 8 represents the relational world. In relational world a software metric for complexity is used to measure the code complexity. In relational world software properties are quantified and therefore mathematical operations can be used to compare properties of different artifacts [24]. For example the complexity of two source code files can be compared by numbers.

6.1 The scope of software metrics

Software measurement encompasses a variety of measurement and prediction activities carried out in software development and production. The most well-known activities listed by Fenton and Pfleeger [25] are:

- Data collection
- Cost and effort estimation
- Productivity measurement
- Quality models and measurement
- Reliability models and measurement
- Performance evolution measurement
- Structural and complexity measurement

In our thesis the focus is on the last bullet point above, that is structural and complexity measurement. Structural and complexity measures are measures for properties of internal product artifacts. Internal properties of product are those that can be measured purely in terms of product, process or resource itself [25]. For example source code, requirements documentation, models and other documentations are parts of the product artifacts that have internal properties. Internal properties can be complexity of code or size of textual requirements. Measures of internal product properties are internal product measures. In this thesis, we use internal product measures and particularly source code measures for risk assessment.
7 RESEARCH FOCUS

The main focus of this research has been the assessment of technical risks in source code delivery. Generally we view software development consisting of four consecutive phases (Figure 9). These phases are requirements specification, software design, verification and validation, and deployment. As Figure 9 illustrates, in every phase of development there are technical risk management activities, consisting of three phases: risk identification, assessment, and mitigation.

In Figure 9 the design phase as well as identification and assessment of technical risks are depicted darker, indicating our research focus. The general research question (RQ) presented in section one (Introduction) is broken down into five RQs presented below (Figure 10).

Part 2 (paper 1):

RQ 1: How can we define technical risk in order to support effective risk assessment?

In this paper we explore technical risks in depth and its difference from other types of risks. We have conducted three workshops with participation of software engineers of collaborating organizations. A set
of technical risks have been identified and presented to software engineers. A definition of technical risk is proposed. In the next step the research has been focused on identifying the drivers of technical risks and monitor.

**Part 3 (paper 2):**

RQ 2: How to monitor code complexity and changes when delivering feature increments to the main code branch

The complexity and changes that are two important drives of technical risks have been explored. Different aspects of complexity and changes have been measured. The evolution of complexity and changes has been investigated over time. As a result of these investigations a measurement system has been developed at Ericsson for monitoring evolution of source code changes and complexity over time. This led to raising a research question concerned with identification and assessment of technical risks with source code delivery.

**Part 4 (paper 3):**

RQ 3: How to effectively identify risky source code and assess the risk, when delivering new feature increments in Agile development?

In this paper a method and supporting tool is suggested for identifying risky source code. Three software complexity and two change metrics have been investigated. A technical risk assessment method has been developed relying on a combination of two software metrics: McCabe’s cyclomatic complexity and number of revisions of a source code file. The method has been evaluated at Ericsson and Volvo GTT. The supporting tool has been integrated with the infrastructure of the development organization at Ericsson for systematic usage. In order to deepen our understanding how different aspects of complexity affect technical risks, the next research question has been formulated.

**Part 5 (paper 4):**

RQ 4: How to reduce code complexity considering various aspects of complexity?

Various complexity aspects of source code on five large software products have been investigated. Two main types of complexity have been delineated for a source code functions. These two types conditionally are called internal and outbound complexities. The investigation of these two aspects of complexities has showed that no matter how large the software is, the product of these two complexity measures for a function cannot exceed a certain limit. We suggested that, the source code functions, which have high complexity, considering both aspects of complexity, should be reviewed and refactored. The evaluation has been carried out at Ericsson and Volvo GTT by a group of software engineers. Concluding the investigation of technical risk
assessment in source code delivery, we addressed a task of developing risk profile in the last paper.

**Part 6 (paper 5):**

*RQ 5: How to profile technical risks in large software development organizations?*

In this paper we give a definition of profiling pre-release product performance. What the profile means and how it is used for organizations is discussed. We show how profiling technical risks can be informative and supportive in decision making for software engineers.

## 8 RESEARCH METHODOLOGY

The presented research relies on action research methodology [26, 27]. The methodology enables close collaboration with companies focusing researchers’ attention rather on “how things are done” in the organizations than “how engineers say things are done”. This implies that researchers are a part of the organization and observe the product and processes themselves.

### 8.1 Action research methodology

“Action research” is a term coined by Kurt Lewin already in 1946 [28] describing a methodological approach for research in social sciences. The methodology is maintained by Susman and Evered [26] and later adjusted to information science by Baskerville and Wood-Harper [27]. The necessity of action research methodology emerged when large numbers of post-war people were examined. Positivistic scientific methodologies turned out insufficient when investigating psychology of a large number of people, because every “case” was considerably different from the other “case”. Lewin applied a clearly defined methodology, where in a patient’s treatment an action was designed and applied for two purposes:

1. Helping the patient for recovery and
2. Adding body of knowledge in the science of social psychology.

Originally, action research had five basic steps [26]:

1. Diagnosing the problem
2. Action planning
3. Action taking
4. Evaluating
5. Specifying learning

Figure 11 presents the general phases of action research. The research cycle starts with diagnosing the problem. In the field of engineering, action research is often conducted due to existing problems in an organization. Therefore, the phase of diagnosing usually converts into formulating research questions of existing problems.

![Diagram of conducting an action research project]

Figure 11 An overview of conducting an action research project

In this phase, the research questions are formulated based on the problems the companies have. In the second step, researchers design methods and tools for both answering the research question and solving the emerged problems. In the fourth step, the researchers study and document the consequences of the conducted action in the organization. In the last step, the researchers specify the learning and communicate it with software engineers.

The researchers usually conduct the research being at company and working with engineers side by side. This way of working permits observing how exactly things are done in the organization and learn the organizational change when a designed action is applied.

In positivist scientific methodologies, such as experimentations, the methods are created in laboratory, where the conditions are strictly controlled Oquist [29]. Then, the developed methods are applied in real life situations to specify the extent on how well the methods meet their requirements. However, as human interests are interwoven and change over evolution of the organization, it is not possible to have strictly controlled experimentation in organizations. Therefore, the
experimental studies fail to create similar environment in laboratory [27], and often the results of such experiments cannot be applied for organizations. Action research provides with an opportunity of designing an action, applying it, and learning the outcome in the organization.

8.2 Application of action research

The philosophy of action research is that every complex system is unique and requires unique approach for solving a problem emerged in it. Studying the complex systems by being a part of them, designing actions, applying actions and studying the consequences is one effective way of quickly gathering body of knowledge about complex systems that are in the same field but are somewhat different.

In our case four organizations have been involved in the research. Two of them have primary role in terms of conducting the research and evaluation the results. The other two have secondary role in the research. We have been a part of the organizations by investigating the formulated problems in the organizations. We have designed methods for solving the problems, and have designed actions based on results. Then we learned the consequences of taken actions and documented it. The process of collecting body of knowledge per company and per research question is illustrated in the Figure 12.

![Figure 12 Action research for generating knowledge on technical risk assessment of code delivery](image_url)

In our research besides collecting body of knowledge we also targeted improving the practice of collaborating organizations. In order to do this we established researcher-engineer collaboration environment. In this environment both engineers and researchers are involved in the research which is called collaborative practice research.
Collaborative practice research allowed us to ease the understanding of obtained results and problem solving in the organizations. We specified similarities and differences between organizations and studied the impact of conducted action on the organization in a long period of time.

**8.3 Methods of data collections**

In this research we have strong focus on observing how “things” are instead of what software engineers say how “things” are. For this reason, indicators of facts and conditions are collected by relying first on analytical metrics and then engineers of reference groups, formed in the organizations. Particularly when focusing on assessing technical risks of source code delivery, source code measures have been collected by automatic tool support. An overview of the collection of measures by tools is visualized in Figure 13. Reference groups consisting of engineers have been involved in periodical meetings in order to discuss and verify the appropriateness of selected measures and correctness of the measurements. By involving engineers of different interests, e.g. engineers, design architects, testers and managers we could get multiple perspectives and interpretations of what the measures showed.

![Figure 13 An overview of data collection procedure](image)

The exact number of involved engineers and their role are described in detail in papers, because in every part of research there were different involvements of engineers.
8.4 Methods of data Analysis

Correlation analyses were used for investigating dependencies of the measures and selecting the measures in an effective way for risk assessment (8.4.1) [31]. Correlograms of measures were developed to visually investigate the dependencies between the measures and separately study the outliers (8.4.2). Time series diagrams have been used for observation of the evolution of measures over development time (8.4.3). Reference groups have been formed for first discussing the appropriateness of the selected measures and then validating drawn conclusions (8.4.4). The information and conclusions extracted from the statistical and analytical operations were communicated with reference groups and discussed for validation.

Figure 14 presents the general overview of the research methods. On the left side of the picture the analytical and statistical tools are presented for analyzing the data and doing inferences. On the right hand the reference group is presented, which helped with interpreting and evaluating the results and provide insights on future research direction.

![Figure 14 Process of data analyses and results in this research](image)

Based on these two different means, methods and tools for technical risk assessment were developed and evaluated.
8.4.1 Correlation Analysis

The correlation analysis was used to identify the dependencies between investigated measures. Correlation analyses are a group of statistical analysis techniques that permits quantifying the linear dependencies between variables. Pearson product-moment correlation coefficient [32] was used for quantification of the measures’ dependencies. Three types of dependencies were investigated in order to make decisions on which metrics to select for risk assessment. These were:

1. Strong positive correlation between two measures. This indicates that the two measures
   a. Characterize the measured entity (source code) from the same perspective. Both measures have similar effect on technical risk. In this case only one of the measures has been selected for risk assessment, as including the other do not add additional predictive power for risk assessment
   b. Have the same cause for magnitude of those measures. The common cause has been identified and studied. Understanding the cause have provided engineers with more knowledge over the product design
2. Strong negative correlation between two measures. This indicates that the two measures
   a. Characterize the measured entity from the same perspective and have opposite effect on technical risk. In this case only one of the measures has been selected for risk assessment, as including the other do not add additional predictive power for risk assessment
   b. Have the same cause for magnitude of those measures.
3. Weak or no correlation between two measures: This indicates that the two measures
   a. Characterize the measured entity from different perspective and have different effect on technical risk. In this case both measures have been used for risk assessment, as their combined predictive power was greater than any of them used alone
   b. Have different causes for magnitude of these two measures
The measures are metrics that describe various aspects of software code, such as complexity, changes, size and dependencies. The number of errors for source code has also been collected as a measure of defect-proneness.

8.4.2 Correlograms

Correlograms are statistical correlation visualization techniques which permit visual observation of dependencies, identifying outliers and observing dependency patterns [33]. A correlogram represents multiple diagrams of correlation plots for several variables in one place. In order to visually observe the dependencies of variables correlograms have been developed. The visual observations have permitted us to detect the outlying data points or sub-groups and investigate why they do not follow the main trend of the data. The reasons and possible explanations have been communicated with reference groups and documented. While correlation coefficients can provide a quantitative number on how strong the dependencies are between variables in average, the correlograms allow local observation of subsets of data.

8.4.3 Time series

We have used time series technique for understanding the reasons that influenced the behavior of software measures over time. Time series is an ordered sequence of values of a variable at equally spaced time intervals [34, 35]. In our case the variables are measures that influence technical risks. By understanding why different measures for particular entities change over time we could investigate that entities and understand the reasons behind the changes. The measured entities have been source code files and functions.

8.4.4 Reference Groups

The reference group is a method in social psychology that serves as standard of comparison for respondents [36]. Since then reference group has been widely used in other fields of qualitative research. In our study reference group was a mean for comparing analytically and statistically obtained results with software engineers’ perception. Two reference groups have been formed: One at Volvo GTT and one at Ericsson, for discussing the results and getting feedback. A part of evaluation is also done by relying on reference groups. A reference group in our case typically has consisted of five to six software engineers including a line manager, a quality manager, design architects and engineers. Software
engineers have been responsible for different development activities, thus had sundry perspectives in interpretations of results, which have allowed us to avoid biased conclusions.

9 VALIDITY EVALUATION
FRAMEWORKS USED IN THE THESIS

In this research we have constructed our research relying on studies by and Susman and Evered [26], and Baskerville and Wood-Harper [27]. Susman and Evered assess the scientific merits of action research and indicate the key conditions that should be held in action research for generating knowledge. These conditions are encapsulated in six characteristics of action research, which we adapted to our research in the following way:

1. Action research is future oriented in dealing with the practical concerns of software engineers. It should permit creating more desirable future for the engineers and organization
2. Action research is collaborative. Independence between researchers and software engineers is essential. The direction of research process should be partly a function of needs and competences of the two
3. Action research implies system development. The research process should encourage the development of the capacity of a system to facilitate, maintain, and regulate the cyclical process of diagnosing, action planning, action taking, evaluating, and specifying learning
4. Action research generates theory grounded in action. The theory should provide a guide for what should be considered in the diagnosis of the organization.
5. Action research is agnostic. The action researcher should recognize that his theories and prescriptions for action are themselves the product of previously taken action, and therefore, are subject for reexamination and reformulation upon entering every new research situation
6. Action research is situational. The action researcher should know that many of the relationship between software engineers, processes and product are a function of the situation as relevant actors currently define it
Baskerville and Wood-Harper propose a set of strategies in order to provide rigor, when conducting action research. We strictly followed these strategies in order to overcome challenges we faced in our research. These strategies were as follows:

1. Considering the appropriateness of action research for answering the research question
2. Establishing a formal research agreement between software engineers and researchers
3. Provisioning a theoretical problem statement
4. Planning the measurement methods to support methodical data collection
5. Maintaining collaboration and subject learning
6. Promoting iterations for repetitive action planning, action taking, and evaluating
7. Restraining generalization of results

According to Baskerville and Wood-Harper, the last point of the enumerated strategies is the most problematic to realize when conducting action research. The researchers and engineers are interested in obtaining generalizable results, however the nature of action research restricts it.

According to Checkland and Holwell \[37\] in order to offset the threat of generalizability one pivotal criterion should be fulfilled. This criterion is the recoverability of the research, which requires the perspicuous representation of research environment and steps of process. This consideration allows the replication of results. Even though the research might not be replicated in the same environment as it has been done originally (and therefore it is not exact replication anymore), the proper documentation of variation points can generate a good insight on produced results. We followed this criterion when conducting our research in different companies.

The aforementioned strategies of action research were adopted in order to construct scientific rigor around our research. Furthermore, all the details of evaluation per action research project are provided in every paper as a separate section of evaluation.
PART 2: DEFINING TECHNICAL RISKS IN SOFTWARE DEVELOPMENT
Abstract

Challenges of technical risk assessment are difficult to address, while its success can benefit software organizations appreciably. Classical definition of risk as a “combination of probability and impact of adverse event” appears not working with technical risk assessment. The main reason of this is the nature of adverse event’s outcome which is rather continuous than discrete. The objective of this study was to scrutinize different aspects of technical risks and provide a definition, which will support effective risk assessment and management in software development organizations. In this study we defined the risk considering the nature of actual risks, emerged in software development. Afterwards, we summarized the software engineers’ view on technical risks as results of three workshops with 15 engineers of four software development companies. The results show that technical risks could be viewed as a combination of uncertainty and magnitude of difference between actual and optimal design of product artifacts and processes. The presented definition is congruent with engineers view on technical risk. It supports risk assessment in a quantitative manner and enables identification of potential product improvement areas.
1 INTRODUCTION

Managing risks of inefficient product design is of great importance for large software development organizations. The general features of inefficient design are well-known: defect-prone or unmaintainable code and models, untestable or untraceable requirements, etc. Chittister and Haimes [38] define the technical risk as the probability and impact of an adverse event. Boehm [39] defines the risk similarly and discuss the top ten risks in software development, most of which are related or directly affect software design. The Software Engineering Institute [4] relies on this definition when outlining the risk management processes. In these studies the definition of risk enables risk quantification by regarding the expected loss as a product of probability and impact of an adverse event. In order to avoid the “probability” element from risk quantification, which is not always possible to estimate explicitly, Barki defines risk as a product of uncertainty and loss [40]. Also, there are others, who view the risk as a qualitative concept that can be estimated subjectively [41, 42]. At present, either classical risk management approaches are applied, which view the risks as a combination of discrete values of probability and impact, or qualitative assessment is performed. However, in practice it is rare to have a discrete value of impact. For example, we cannot assume that a software program either will have an error or will not, because the program might have one, ten or 60 errors, and one error might be acceptable for a product. Similarly we cannot expect a piece of code to be either maintainable or unmaintainable, a requirement to be either traceable or not traceable, etc.

The presence of a continuous component in the risk concept disables quantification of risk exposure as a product of probability and impact of an adverse event. As a result no comprehensive definition of technical risk appears to exist in the field of software engineering, which would permit an effective risk assessment. Therefore an open question remains:

*How can we define technical risk in order to support effective risk assessment?*

The aim of this paper is to explore the essence of technical risk and define it in a manner that it supports risk assessment and management.

In this study we identified a list of technical risks based on input from four large software development companies: Ericsson, Volvo Car Corporation (CC), Volvo Group Truck Technology (GTT) and Saab. We show that the uncertainty on the difference between actual and optimal
Defining technical risk in software development

designs of product is a key indicator of technical risks: Ones this difference is identified the risk converts into a problem. Provided new definition of technical risk uses the concepts of uncertainty and difference from optimal design as two continuous measures. Mitigation of such risks becomes a task of identifying and reducing the potentially inefficient design decisions.

The remainder of this paper consists six sections. First we introduce the motivation of this study. Then the definition of technical risk is presented. In section four a list of technical risks, identified in the four companies, are discussed. Section five proposes a risk forecast model for risk assessment. Subsequently we give a brief overview of related studies, and make concluding remarks.

2 MOTIVATION OF THE STUDY

In our previous research which was initiated by Ericsson, we were requested to create a method for identifying risks associated with source code delivery. That is, to create a method for identifying the most defect-prone or hard-to-maintain source code before delivery. The results of research were also applied in Volvo GTT and Saab to identify risky source code [43]. Approximately a year after, we conducted research at Volvo GTT to do similar analysis with textual requirements. The purpose was to identify risky requirements in early development phases in order to permit early reviews and improvements. From the companies’ side there is an increasing need for conducting similar analysis on such aspects of development as “identifying risky Simulink models” and “risk based test selection”. However, ones the research question was formed, it was unclear how the risk should be regarded. How can we clearly state what is a risk and if the risk occurs what is the impact or the loss? If the risk is viewed as a probability and impact of adverse event it will not be possible to define the impact explicitly, since in practice it does not have a fixed value. For example maintainability, readability and fault-proneness of source code, traceability, feasibility and ambiguity of requirements, efficiency and effectiveness of executed test cases are properties of software artifacts the impact of which are not fixed values. These properties are comparative, which means that a requirement can be more or less traceable, a source code function can be more or less defect-prone etc. As there is no explicit value for the impact, the probability of having that impact cannot be defined also. Particularly in our research the risk of low maintainability and high defect-proneness could not be viewed as a risk that has distinct probability and fixed
Defining technical risk in software development

impact. This kind of risk could be assessed rather by predicting the possible level of maintainability and defect-proneness of the code. One way of doing this could be so: if majority of experienced engineers have a strong feeling, that the code consumes twice more maintenance time than they expect, then there is a tangible risk that the organization might spend significant amount of cost over long run of development. Quantitatively assessing different levels of maintainability and defect-proneness of the code is not an easy task, but its success can determine the effective applications of proactive decisions.

The presence and increasing role of technical risk assessment in software development, as well as difficulties of adopting classical risk definition for technical risks of software development in practice led us to define the risk in a manner that supports technical risk identification, assessment and mitigation.

3 DEFINING TECHNICAL RISKS

In various fields the risk is defined differently. In finance the risk is usually considered as a combination of the probability and the variance of the actual and expected return. In health care, the risk is viewed as a combination of probability and damage. In ISO 31000 the risk is defined as an effect of uncertainty on objectives \[14\]. All of these definitions contain events which ultimately might have an impact on a person or organization (group of people) that have targeted a specific objective to achieve. In certain definitions people or organizations are not mentioned explicitly, but they are implicitly considered. Generally the risk is defined for a particular person or a group of people and it can influence a particular objective that is targeted to be achieved. In software development organizations technical risks can be described as possibility of undesirable events which ultimately affect software engineers and software development organizations. These risks usually accompany various process or product design decisions and can affect processes and product artifacts.

Prior to defining the technical risk we emphasize three pivotal concepts which will be underlaid in our risk definition:

1. The objective that a person or an organization want to achieve
2. The person or organization that the risk emerges for
3. And their decisions or solutions which can change the impact of risk

The first concept stresses the fact that a certain objective can be influenced by the risk. The second concept indicates that the objective is
defined for a person or an organization. If there are more than one interrelated persons or organizations the same risk can affect them differently: Defining the risk from one particular person’s or organization’s perspective creates a possibility of a situation, where reducing the risk exposure for one person might cause an increase of risk exposure for the other person. This might happen because the two persons have different objectives to achieve, and the reduction of the same risk for them might mean making contradicte decisions. The third concept is the taken design decision (solution) which the organization implements in order to develop a particular artifact of the product. It is important to notice that the risk is usually associated with a decision where multiple scenarios are possible. When encountering such situations, it is not always explicit which decision is optimal for a particular person or organization. Moreover, the more uncertainty there is associated with decisions, the more unpredictable the outcomes of those decisions are. In the presence of complete uncertainty there is no way of determining which decision is preferable. While there is always an actual design decision for a particular development operation, we assume that there is always an existing optimal decision for that operation. Furthermore, we consider that for any decision there is a technical risk associated with it. If the taken decision is an optimal solution than the risk exposure is minimal. Making no decision is also a decision that is, leaving the current condition as it is.

Considering that the risk is about possibility of suffering loss in whatever design decision (solution) the organization makes, the impact of the risk can be defined as the magnitude of difference between actual and optimal design solutions. As long as this difference is not known, the organization takes on a technical risk. The more uncertainty is associated on how much this difference is the more risky the situation is. But ones the organization knows this difference precisely, the risk converts into a problem. By this understanding of risks, for a given person or organization and for a given development operation, when there are several possible design solutions to achieve an objective, we define the risk as:

The technical risk is the degree of uncertainty on the magnitude of difference between the actual and optimal design solutions.

We consider that this magnitude of difference is measured either by internal metrics of software quality [44], time or cost of design. The advantage of this definition is that there is a target level of risk minimization which can be achieved by a specific design solution. From this standpoint, addressing technical risk management means to identify the magnitude of difference between optimal and current solutions, and
Defining technical risk in software development

minimize that difference as much as possible. The magnitude of difference can be viewed as the loss in classic definition of risk. In practice, however, the optimal design solution cannot be determined precisely. Advanced methods and measures might be applied for determining the most preferable design of an artifact, which can be considered as an optimal design.

A symbolic visualization of risk exposure is illustrated in Figure 15. The figure illustrates the concept of technical risk as a combination of the uncertainty of the actual non-optimal design solution and the unknown unnecessary cost generated as a result of that solution (dark grey area of the figure). When the deficiency of the design solution is identified, the organization has a known design problem (lower-right square). When the current solution is the optimal one but the organization is uncertain about it, it means there is a problem of insufficient assessment, which should show how well the product artifact or process is designed (upper left square). In practice this case is rather rare. If there is an optimal solution and the organization is certain about it by relying on well-established assessment methods, then there is an opportunity of gaining minimal development cost and reusing the current design solutions in the product design.

![Figure 15 The technical risk as uncertainty on inadequate design](image)

The arch-arrow in the Figure 15 shows the transaction path from risk to opportunity. Ideally the organization pursues the transaction from risk to opportunity directly but in practice this happens by first identifying the non-optimal design solutions (moving to the low-right area) and then
Defining technical risk in software development

redesigning them (moving to the low-left area). The aforementioned explanation outlines the main characteristic of technical risk: It is the non-optimal design solution of process or product, which the organization is unaware of. In next section we present a list of risks identified at four companies and discuss them.

4 SOFTWARE ENGINEERS VIEW ON TECHNICAL RISKS

We discussed technical risks of software development with engineers, architects and line managers of large software development organizations in four companies. We organized two company specific workshops on technical risk identification at Volvo GTT and at Ericsson. Subsequently one final workshop was organized with all four companies to identify the main list of technical risks that engineers face in software development.

At Ericsson workshops had a specific focus on risks associated with developed source code. At Volvo GTT the purpose of the workshops was identifying main risks specifically associated with requirements implementation and delivery.

The final workshop with all companies was meant to harmonize previously identified technical risks and complement with new ones if there are. The list of all risks, that were identified during the workshops at Ericsson and Volvo and was crystalized during the last workshop with four companies, is presented in Table 1. In the first column of the table we have registered the main description of risks. The risks that are written with bold text are previously identified by other researchers [45, 46]. The risks that are written in italic texts are similar to risks found by other researchers but are not exactly the same.

The descriptions of risks in Table 1 imply that the possible outcomes of these risks are not two discrete values, that is, either the described adverse event will happen or not. All of risks in the table are decisively dependent on product or process design solutions. The better the design solutions are the little the risk is.

Table 1 Main technical risks identified at four companies

<table>
<thead>
<tr>
<th>N</th>
<th>Risk</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Multiple “wishes” in one requirement</td>
<td>This requirement contains multiple requests which might make it difficult to develop and test</td>
</tr>
<tr>
<td>2</td>
<td>Inappropriate</td>
<td>This requirement contains pseudo-code,</td>
</tr>
<tr>
<td><strong>representation of requirements</strong></td>
<td>references to other documents or requirements, etc. which might decrease its understandability from semantic point of view</td>
<td></td>
</tr>
<tr>
<td>----------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td><strong>3 Untraceable requirements</strong></td>
<td>This requirement has extensive coupling with other requirements which might make it vulnerable towards outer factors. Late changes of this requirements is highly likely</td>
<td></td>
</tr>
<tr>
<td><strong>4 Notes and assumptions in requirements</strong></td>
<td>This requirement contain notes and assumptions which might result in developing wrong functionality</td>
<td></td>
</tr>
<tr>
<td><strong>5 Unfeasible, unclear or untestable requirements</strong></td>
<td>High likelihood that this requirement is hard to understand and implement because of unclear syntactic description</td>
<td></td>
</tr>
<tr>
<td><strong>6 Requirements’ changes</strong></td>
<td>This requirement is changed often because of strong business objectives but it might cause risk of late product delivery</td>
<td></td>
</tr>
<tr>
<td><strong>7 Adding requirements in late phases</strong></td>
<td>This requirement is added after “freezing” the requirements which might delay the product delivery</td>
<td></td>
</tr>
<tr>
<td><strong>8 Inadequate architectural solutions</strong></td>
<td>This component has unwanted hidden dependencies which might create maintainability and real time performance issues</td>
<td></td>
</tr>
<tr>
<td><strong>9 Simple design mistakes</strong></td>
<td>For some unclear reasons engineers often make simple mistakes in this file. The reasons might be lack of cohesion, non-commented code and complexity. This causes many errors in that file</td>
<td></td>
</tr>
<tr>
<td><strong>10 Unrealistic time and cost estimates</strong></td>
<td>Because of semantic complexity engineers often underestimate the effort in this file. It creates additional efforts for reconsidering the development plans</td>
<td></td>
</tr>
<tr>
<td><strong>11 Gold plating</strong></td>
<td>This file is a result of an over-flexible design which makes the file difficult-to-maintain</td>
<td></td>
</tr>
<tr>
<td><strong>12 Dependence on external supplied components</strong></td>
<td>This file is directly and “heavily” dependent on components outside of our control. It might cause late errors</td>
<td></td>
</tr>
<tr>
<td><strong>13 Dependency on other tasks</strong></td>
<td>This file is directly and &quot;heavily&quot; dependent on execution of tasks outside of its control. It might cause late errors</td>
<td></td>
</tr>
<tr>
<td><strong>14 Real-time performance shortfalls</strong></td>
<td>This file is vulnerable to real-time performance of the system. There is high likelihood to get post-delivery error reports from the customers</td>
<td></td>
</tr>
<tr>
<td><strong>15 Defect-prone, unmaintainable or unmanageable code</strong></td>
<td>This file is complex and non-cohesive which might increase its defect-proneness and decrease the maintainability</td>
<td></td>
</tr>
<tr>
<td><strong>16 Insufficient or</strong></td>
<td>This test is insufficient. Many post-delivery</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td><strong>ineffective testing</strong></td>
<td>defects might be reported</td>
<td></td>
</tr>
<tr>
<td><strong>17</strong> Low number of builds on the integration branch</td>
<td>Low number of builds indicates high likelihood that after integration many defects might appear and impede the smooth delivery</td>
<td></td>
</tr>
<tr>
<td><strong>18</strong> Accessing test objects</td>
<td>Missing test objects might create development disruption</td>
<td></td>
</tr>
<tr>
<td><strong>19</strong> Queuing</td>
<td>Several processes might be waiting for the completion of this particular process which seems to be delayed.</td>
<td></td>
</tr>
<tr>
<td><strong>20</strong> <em>Underestimating the delivery</em></td>
<td>time between “ready” and actual integration of build in main code branch seems to be underestimated which can create high likelihood of late delivery</td>
<td></td>
</tr>
<tr>
<td><strong>21</strong> Mismanaging software variants</td>
<td>The likely non-optimal separation or combination of software variants for different platforms creates a risk of additional cost</td>
<td></td>
</tr>
<tr>
<td><strong>22</strong> Defect turnaround time’s underestimation</td>
<td>Underestimation of defect turnaround time creates high likelihood for unsatisfied expectations of customers</td>
<td></td>
</tr>
<tr>
<td><strong>23</strong> Splitting software into parallel releases</td>
<td>Parallel releases of software might create unexpected additional cost</td>
<td></td>
</tr>
<tr>
<td><strong>24</strong> <em>Missing key-specialist in development</em></td>
<td>Certain development process is heavily dependent on a key-specialist the absence of whom can create disruption of development and queuing</td>
<td></td>
</tr>
</tbody>
</table>

For example the source code cannot be either maintainable or not. Instead, the maintainability can be regarded as more or less maintainable. If we quantify the probable loss as consequence of these risk, we get an interval instead of a fixed value, because the more maintainable the code is the less the loss is. Other examples are: (i) increasing changes of requirements causes increasing changes and late errors in a file, (ii) increasing number of “wishes” in one requirement causes decreasing testability and feasibility and (iii) longer queuing time causes higher development cost.

For several risks there is no explicit mitigation strategy that can reduce the risk. These risks contain high level of uncertainty for decision making. For example according to engineers, the over-flexible design of the file triggers a risk of undesirable growth of size and complexity, nonetheless, it does not mean that the simplest design minimizes the risk. Practically, at some point the simplification of this file’s design might trigger the increase of risk again, because the defect-proneness of
the file might increase due to its inability of operating in variety of conditions. Another example is: Volvo GTT produces similar electronic control units for several trucks (Renault, Volvo, Mack, etc.). The question is, should these variants be developed with having completely different requirements design and source code or they can have common design with defined variation points? How this variation points can be determined and how much the development can be carried out jointly or separately? On the one extreme, if everything is separate, the organization might run into big amount of additional costs, because of extensive number of requirements and code that is not reused for all software variants. On the other extreme, there is likelihood that certain part of code and requirements that are different for truck variants, are reused and defects can be left in the software. In this types of risks trade-offs between multiple design choices should be considered.

The risks listed in the table are dependent on design solutions of particular process or product artifact. The design-oriented nature of risks implies that in order to mitigate the risk there is a need of optimal design solutions. In other words the minimal risk exposure can be achieved when optimal solutions are applied. In the next section we discuss the relevance of risk forecast models by application of measurable properties of software product or processes.

5 RISK ASSESSMENT BY FORECAST MODEL

We propose that technical risks can be assessed by risk forecast model which relies on measurable characteristics of software design and processes. The use of software measures as risk predictors are widely suggested [8, 47, 48] and software organizations that we collaborated with, where using various measures as technical risk predictors.

Figure 16 shows that the risk has consequences, which can be measured and which the organization would like to minimize. The risk also has indicating characteristics which can be measured for risk assessment. The task is to develop such a mathematical model that using measures of product properties can forecast the consequences of risk. In many cases it is not possible to forecast the consequences precisely as a single number for quality or cost, but an interval of loss can be predicted by certain confidence level. In our previous research [43] for Ericsson we developed a measurement system, which predicted the risk of defect-proneness and maintainability of source code.
Defining technical risk in software development

The representation of our measurement system by risk forecast model is illustrated in Figure 17. As the figure shows the consequences of risk are additional maintenance time and defect-proneness of files. The software engineers at Ericsson had a perception that the maintenance time is higher and the source files are more defect-prone than expected. This means that there was initial information about non-optimal design of source code but it was not thoroughly known.

In order to assess the risk we identified the risk-indicating characteristics, which were code complexity and change frequency. Complexity and change frequency were measured by calculating cyclomatic complexity (M) and number of revisions (NR) of source code files. Then we designed a model and measurement system based on these measures to assess the risk. Ones the files were assessed, the level of uncertainty on previous design decision of files sharply dropped, revealing the existing problems. Afterwards the engineers made decisions of resolving the problem.

In practice, however, there are noticeable problems with assessing what the best design of source code is and how the current design differs from that of optimal design. The identification of optimal design seems to be dependent on more parameters of source code than we are able to measure currently. In foregoing example the superposition (combined metric) of number of revisions and cyclomatic complexity can give an
Defining technical risk in software development

insight how well a source code file is designed, but that combined metric by no means gives a direct estimate on the magnitude of difference between optimal and actual designs of source code file. In fact, what this combined number gives is an estimate how well this particular file is designed compared with other files in the system. We know that from certain point the increasing cyclomatic number and increasing number of revisions indicate decreasing code quality. However, this certain point cannot be precisely understood as it can be different for different files. This difference arises from the limitedness of cyclomatic complexity and number of revisions as measures.

6 RELATED WORK

A comprehensive definition of risk can be found in Kaplan’s and Garrick’s work [15], where the risk is defined as a superposition of uncertainty and loss. They claim that the risk as a product of probability and consequence is misleading. Furthermore, the authors show that the risk cannot be represented by a single number but rather with a curve, where the risk is not a point on that curve as an expected value but the curve itself.

The term technical debt coined by Cunningham [49] is used to discuss the cost that organization pays over time due to non-optimal design of product artifacts. Kruchten, et al. [50] solidifies this term in the context of whole software engineering and define it as “postponed opportunities” or “poorly managed technical risks”. This view is a support for our work, or we may say, our work is harmonious with this view because the difference between actual and optimal design is viewed as the main composite part of the technical risk. Chawan, et al. [51] notice that technical risks emerge because of excessive constraints on the development and poorly defined parameters or dependencies in the organization. Ropponen and Lyytinen [52] conclude that much of the time project risks are not well-understood and their effective assessment and management should be carried out by experienced and well educated managers. We believe that our new definition of risk provides a tangible insight to technical risks so it could be better understood. Bannerman [53] found that the risk management procedures, defined in the literature, do not necessarily support the risk management activities in practice. Bannerman observed that in software engineering on one hand the risk management research lags the needs of practice and on the other hand risk management in practice does not adhere to the research.
The current study can be considered as a step for linking general concepts of software risk with its practical needs of management.

7 CONCLUSIONS

In software development organizations continuous identification and assessment of technical risks is an essential activity. The consequences of technical risks can include overall increase of development time and cost, and decrease of pre-delivery product quality. Despite the importance of technical risk assessment, the existing definitions of risks appear not supporting risk assessment. The reason is that in software development technical risks mostly emerge due to inadequate design solutions of processes and product artifacts, but these solutions cannot be regarded as explicit adverse events.

In this research we outlined the essence and main characteristics of technical risks in software development. We showed that while in other fields the risk can be viewed as a product of probability and loss, in software development an explicit occurrence of an adverse event does not exist thus a probability for that occurrence cannot be assessed. Instead we regarded the loss as a continuous variable which is strongly dependent on how good the design solutions of product and processes are. Bigger difference of actual and optimal design solutions and higher uncertainty associated with actual design indicates higher risk exposure. This view of technical risk facilitates the formulation of risk mitigation strategy: (i) identification of optimal design prerequisites and (ii) redesign of product artifacts.

Acknowledgment

We would like to express our thanks to the companies for their support in the study. This research has been carried out in the Software Center, Chalmers, University of Gothenburg, Ericsson, Volvo Group Truck Technology, Volvo Car Corporation and Saab.
PART 3: MONITORING EVOLUTION OF CODE COMPLEXITY AND MAGNITUDE OF CHANGES
Abstract

Complexity management has become a crucial activity in continuous software development. While the overall perceived complexity of a product grows rather insignificantly, the small units, such as functions and files, can have noticeable complexity growth with every increment of product features. This kind of evolution triggers risks of escalating fault-proneness and deteriorating maintainability. The goal of this research was to develop a measurement system which enables effective monitoring of complexity evolution. An action research has been conducted in two large software development organizations. We have measured three complexity and two change metrics of code for two large industrial products. The complexity growth has been 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 potentially problematic files and functions for manual assessment and improvement. A measurement system was developed at Ericsson to support the monitoring process.
1 INTRODUCTION

Actively managing software complexity has become an important aspect 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 quality of software [5, 54, 55]. The continuous increase of code base and growing complexity can lead to large, virtually unmaintainable source code if left unmanaged.

A number of metrics have been suggested to measure various aspects of software complexity and evolution over development time [56]. Those metrics has been accompanied with a number of studies indicating how adequately the proposed metrics relate to software quality [31, 57]. Complexity and change metrics have been used extensively in recent years for assessing the maintainability and fault-proneness of software code [58]. 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 to monitor code complexity and changes effectively when delivering feature increments to the main code branch?

The aim of this research was to develop method and tool support for actively monitoring complexity evolution and drawing the attention of industries' software engineers to the potentially problematic 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 a case study at 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 draws attention of the self-organized Agile teams to a handful of functions and files. These functions and files are manually assessed, and
the team formulates decisions before the delivery on whether they can cause problems.

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 [59], in the context of market driven software development. The metrics presented by Fritz measure such aspects as continuous integration as pace of delivery of features to the customers. These metrics 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 [60]. 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 [61] 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 computerized 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. concept of measurement errors, automation, etc. An example of a similar measurement system is presented by Wisell [62], 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 [63]) 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 [64] present a generic way of modeling measures and building more advanced measures from less complex
Monitoring evolution of code complexity and magnitude of changes

ones. Their work is linked to the TychoMetric tool. The tool is a very powerful measurement system framework, which has many advanced features not present in our framework (e.g. advanced ways of combining metrics). A similar approach to the TychoMetric’s way of using metrics was presented by Garcia, et al. [65]. Despite their complexity, both the TychoMetric tool and Garcia’s approach can be seen as alternatives in the context of advanced data presentation or advanced statistical analysis over time. Our research is a complement to [64] and [65]. 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 [66, pp. 99-122] claims that the need for customized measurement systems for teams is one of the most important aspects in the adoption of metrics 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 [67] in the domain of software development projects.

3 DESIGN OF THE STUDY

This case study was conducted using action research approach [68-70]. The researchers were part of the company’s operations and worked directly with product development units. The role of Ericsson in the study was the development of the method and its initial evaluation, whereas the role of Volvo GTT was to evaluate the method in a new context.

3.1 Studied Organizations

**Ericsson**: The organization and the project within Ericsson developed large products for mobile packet core network. The number of the developers in the projects was up to a few hundreds. Projects were executed according to the principles of Agile software development and Lean production system, referred to as Streamline development within Ericsson [71]. 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, testing, etc.
Volvo GTT: The organization which we worked with 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 our study we analyzed two different products – software for a telecom product at Ericsson and software for two ECUs at Volvo GTT.

Ericsson: The product was a large telecommunication product composed 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 has been 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 several 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, that is, running the measurement system whenever feedback was needed for engineers.

3.3 Reference Group

During this study we had the 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 on a bi-weekly meeting basis 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 2 presents the complexity measures, change measures and deltas of complexity measures over time. The definitions of measures and their deltas are provided also.

Table 2 Metrics and their definitions

<table>
<thead>
<tr>
<th>Complexity Measures</th>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>McCabe’s cyclomatic complexity of a function</td>
<td>M</td>
<td>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’, ‘&amp;&amp;’, ‘</td>
</tr>
<tr>
<td>Structural Fan-out</td>
<td>Fan-out</td>
<td>The number of invocations of functions found in a specified function</td>
</tr>
<tr>
<td>Maximum Block Depth</td>
<td>MBD</td>
<td>The maximum level of nesting found in a function</td>
</tr>
<tr>
<td>McCabe’s cyclomatic complexity of a file</td>
<td>M_f</td>
<td>The sum of all functions’ M in a file</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Change Measures</th>
<th>Abbrev.</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of revisions of a file</td>
<td>NR</td>
<td>The number of check-ins of files in a specified code integration branch and its all sub-branches in a specified time interval</td>
</tr>
<tr>
<td>Number of engineers of a file</td>
<td>ND</td>
<td>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</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Deltas of Complexity Measures</th>
<th>Abbrev.</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complexity deltas of a function</td>
<td>ΔM</td>
<td>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 (if available) and function name in order to identify the same function and calculate its complexity change over releases.</td>
</tr>
<tr>
<td></td>
<td>ΔFan-out</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ΔMBD</td>
<td></td>
</tr>
</tbody>
</table>

3.5 Research Method

According to the principles of action research we adjusted the process of our research with the operations of the company. We conducted the study according to the following pre-defined process:

- Obtain access to the source code of the products and their different releases
- Calculate complexity measures of all functions and change measures of all files in the code
• Calculate the complexity deltas of all functions through five releases of both products
• Sort the functions by complexity delta through five releases
• Identify possible patterns of complexity change
• Identify drivers for complexity changes for functions with functions having highest overall delta
• Correlate measures to explore their dependencies and select measures for monitoring complexity and changes
• Develop a measurement system (according to ISO 15939) for monitoring complexity and changes
• Monitor and evaluate the measurement system for five weeks
• The overall complexity change of function is calculated by: Overall delta = |ΔM_{rel,1-2}| + |ΔM_{rel,2-3}| + |ΔM_{rel,3-4}| + |ΔM_{rel,4-5}|
  where |ΔM_{rel,i,j}| is the absolute 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 chose 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.

1. Group 1 - Functions that are newly created and become complex in current release and functions that existed but disappeared in current release.
2. Group 2 - Functions that are re-implemented in current release.
3. Group 3 - Functions that have significant change of complexity between two releases due to development or maintenance.
4. Group 4 - Test functions, which are regularly generated, destroyed and regenerated for unit testing.
5. Group 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 modification.

Figure 18 shows the cyclomatic complexity evolution of top 200 functions through five releases of products. Each line on the figure represents a C function.

In Figure 18 re-implemented functions are outlined by elliptic and old ones by round lines. In reality the number of re-implemented functions are small (about 1 %), however considering the big magnitude of complexity change of them, many of them ended-up in the top 200 functions in the picture, giving an impression that they are relatively many. Figure 19 similarly presents the evolution of Fan-out in the products. Group 3 functions are outlined by elliptic line in Figure 19.
Monitoring evolution of code complexity and magnitude of changes

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 5 functions were unit test implementations. These functions were destroyed and regenerated frequently in order to update running unit tests. Figure 20 presents the MBD evolution of products. As nesting depth of blocks can be relatively shallow, many lines in Figure 20 overlap each other thus creating an impression that there are few functions. We observed that functions in group 1, ones were created, stayed complex over time. These functions are outlined with a rectangular line in Figure 20.

The statistics of functions of all groups are represented in Table 3. 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 relationship between the relation of different groups of functions in the table and in the figures as the figures contains only top 200 functions.
We observed the change of complexity for both long time intervals (between releases) and for short time intervals (between weeks). Figure 21 shows how the complexity of functions changes over weeks. The initial complexity of functions is provided under column M in the figure.

We can see the week numbers on the top of the columns, and every column shows the complexity growth of 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 functions fluctuates irregularly was interesting for the engineers, 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 21. 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 the number of functions that have complexity increase ΔM > 10 can vary between 5-10 while overall number of functions reaches a few tens of thousands in the product.
4.2 Correlation analyses

The correlation analyses of measures were conducted in order to eliminate dependent measures and select a minimal amount of measures for monitoring. The correlation analysis results of complexity measures for the two software products are presented in Table 4. The visual presentation of the relationship of complexity measures is presented in Figure 22. As the table illustrates there is a strong correlation between M and Fan-out for the telecom product and M and MBD for the automotive product. There is a moderate correlation between M and MBD for the telecom product. Generally engineers of reference group concluded that monitoring the cyclomatic complexity among all complexity measures is good enough as there was a moderate or strong correlation between three complexity measures.

<table>
<thead>
<tr>
<th>Telecom / Automotive</th>
<th>MBD</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>0.41 / 0.69</td>
<td></td>
</tr>
<tr>
<td>Fan-out</td>
<td>0.34 / 0.20</td>
<td>0.76 / 0.26</td>
</tr>
</tbody>
</table>

M was chosen because of two reasons:

- 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.
- Fan-out seemed to be a weaker indicator of complexity than M because it rather showed the vulnerability of a function towards other functions that are in that function.

Considering aforementioned conclusions M was chosen among complexity measures to be monitored.

Figure 22 Correlogram of complexity measures
NR and ND are measures that indicate the magnitude of changes. Previously a few studies have shown that change metrics are good indicators of problematic areas of code, as observed by Shihab, et al. [72]. The measurement entity of NR and ND is a file. Therefore in order to understand how change measures correlates to complexity we decided to define the M measure for files (Table 2). Table 5 presents the correlation analysis results for ND, NR and M_f measures.

**Table 5 Correlation of change and complexity measures**

<table>
<thead>
<tr>
<th>Ericsson / Volvo</th>
<th>M_f</th>
<th>ND</th>
</tr>
</thead>
<tbody>
<tr>
<td>ND</td>
<td>0.40 / 0.37</td>
<td></td>
</tr>
<tr>
<td>NR</td>
<td>0.46 / 0.72</td>
<td>0.92 / 0.41</td>
</tr>
</tbody>
</table>

An important observation was the strong correlation between the number of engineers and the number of revisions for the telecom product Table 5. At the beginning of this study the engineers of the reference group at Ericsson believed that a developer of a file might check-in and check-out the file several times which probably is not a problem. The real problem, they thought, could be when many engineers 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 23).

In case of automotive product correlation of 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. Moderate correlation between M_f and NR for the telecom product indicates that complex files are prone to changes. There are always simple files that are changed often due to development.
Considering the correlation analysis 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

Based on the results that we obtained from investigation of complexity evolution and correlation analyses, we designed two indicators based on M and NR measures. These indicators capture the increase of functions’ complexity and highlight the files with highest change magnitude over time. These indicators were designed according to ISO/IEC 15959. The design of complexity indicator is presented in Table 6.

**Table 6 Measurement system design based on ISO/IEC 15939 standard**

<table>
<thead>
<tr>
<th>Information Need</th>
<th>Monitor cyclomatic complexity evolution over development time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measurable Concept</td>
<td>Complexity change of delivered source code</td>
</tr>
<tr>
<td>Entity</td>
<td>Source code function</td>
</tr>
<tr>
<td>Attribute</td>
<td>Complexity of C functions</td>
</tr>
<tr>
<td>Base Measures</td>
<td>McCabe’s Cyclomatic complexity number of C functions – M</td>
</tr>
<tr>
<td>Measurement Method</td>
<td>Count cyclomatic number per C function according to the algorithm in CCCC tool</td>
</tr>
<tr>
<td>Type of measurement method</td>
<td>Objective</td>
</tr>
<tr>
<td>Scale</td>
<td>Positive integers</td>
</tr>
<tr>
<td>Unit of measurement</td>
<td>Execution paths over the C/C++ function</td>
</tr>
<tr>
<td>Derived Measure</td>
<td>The growth of cyclomatic complexity number of a C function in one week development time period</td>
</tr>
<tr>
<td>Derived Measure Function</td>
<td>Subtract old cyclomatic number of a function from new one: ( \Delta M = M(\text{week}<em>i) - M(\text{week}</em>{i-1}) )</td>
</tr>
<tr>
<td>Indicator</td>
<td>Complexity growth: <em>The number of functions that exceeded McCabe complexity of 20 during the last week</em></td>
</tr>
<tr>
<td>Model</td>
<td>Calculate the number of functions that exceeded cyclomatic number 20 during last week development period</td>
</tr>
<tr>
<td>--------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Decision Criteria</td>
<td>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</td>
</tr>
</tbody>
</table>

The other indicator based on NR is defined in the same way: the files that had NR > 20 during last week development time period should be identified and reviewed.

The measurement system was provided as a gadget with the necessary information updated on a weekly basis (Figure 24). The measurement system relies on a previous study carried out at Ericsson [73, 74].

For instance the total number of files with more than 20 revisions since last week is 5 (Figure 24). 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.

We visualized the NR and ΔM measures using tables as depicted in Figure 24.

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

![No of Complex functions](image.png)

**Figure 24 Information product for monitoring ΔM and NR metrics over time**
5 THREATS TO VALIDITY

The main external validity threat is the fact that our results come for an action research. However, since two companies from different domains (telecom and automotive) were involved, we believe that the results can be generalized to more contexts than just one specific type of software development.

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 limit 20 for cyclomatic number established as a threshold in this study 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 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' perception of how fault-prone and unmaintainable the delivered code is.

6 CONCLUSIONS

In continuous software development quick feedbacks on developed code complexity is crucial. With small software increments there is a risk that the complexity of units of code can grow to an unmanageable level.

In this paper we explored how complexity evolves, by studying two software products – one telecom product at Ericsson and one automotive
Monitoring evolution of code complexity and magnitude of changes

product at Volvo GTT. We identified that in short periods of time a few out of tens of thousands functions have significant complexity increase. We also concluded that the self-organized teams should be able to make the final assessment whether the “potentially” problematic is indeed problematic.

By analyzing correlations between three complexity and two change metrics we concluded that it is enough to use two measures, McCabe complexity and number of revisions, to draw attention of the teams to potentially problematic code for review and improvement.

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

In our further work we intend to study how the teams formulate the decisions and monitor their implementation.

Acknowledgment

The authors thank the companies for their support in the study. This research has been carried out in the Software Centre, Chalmers, University of Gothenburg and Ericsson, Volvo Group Truck Technology.
Monitoring evolution of code complexity and magnitude of changes
PART 4: IDENTIFYING RISKY AREAS OF SOFTWARE CODE IN AGILE SOFTWARE DEVELOPMENT
Abstract
Modern software development relies on incremental delivery to facilitate quick response to customers’ requests. In this dynamic environment the continuous modifications of software code can cause 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 code in Agile environment. We have conducted an action research project in two large companies, Ericsson AB 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 metrics, complexity and revisions of a source code file, 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

Growing complexity of modern software products today has become a well-known problem. Escalating fault-proneness and declining maintainability of software are the main risks behind this kind of growth. Due to increasing size of software products and the need for increased development velocity the traditional risk assessment methods [39, 75-79] 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 product engineers in systematic improvement of code quality and mitigation of risks.

Several studies have shown that the code is continuously becoming more complex if left unmanaged [5, 54, 55], and with growing complexity momentous technical risks emerge. Fenton and Neil [8] 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 [80-87]. 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 to 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, controlling different variants of code for different customer groups etc.

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 AB and Volvo Group Trucks Technology (Volvo GTT).

We created a method and supporting tool for identification of risky files. The method is based on 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 drawing the attention of the organization and trigger refactoring initiatives.

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 25 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 the code to any component of the product.

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 [88].

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-time periods. This continuous development with small increments means that working with technical risks, triggered by complexity development, size development and related aspects, also needs to be a continuous activity.
3 STUDY DESIGN

We applied the action research method in our study, that is, close collaboration with industry professionals and regular presence at 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 principles of Agile software development and Lean production procedures, called Streamline development within Ericsson [71]. 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 metrics organization was initiated (7 years before writing of this paper) for calculating and presenting variety of predictors and early warning systems. The developed unit was a large telecom product which constitutes a few million lines of code with several thousands of C/C++ files. The product is released a few times per year with support of service releases. Rational ClearCase serves as version control system by which all the source code of the product is handled. The product has been in development for more than 15 years.

At Volvo Group Trucks Technology (GTT), the organization in which we have worked develops software of Electronic Control Units’ (ECU) for Volvo, Renault, UD Trucks and Mack. Our collaborating unit develops 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 is released in every 6-8 weeks. About 50 designers, business analysts and testers comprise 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

In our study we defined five main research cycles to be carried out:

1. Identification of metrics: Shortlisting a number of published metrics which can theoretically be used in risk assessment.
2. Measurement and analyses: Collecting data for these metrics and analyzing inter-dependencies of the data set.
4. Evaluation of the method with engineers: Evaluating the method over a number of weeks through weekly meetings with reference group, and according to empirical metric validation principles [89].
5. Refinement and evaluation of method in projects: Disseminating the method to all engineers in the project and monitoring the level of 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:

- Obtain access to the source code of the products and their different releases: Decided upon one product per company, releases of the product, decide whether service releases should be included or not
- Set up necessary tools for extracting data: Develop scripts for data collection in Ruby, MS Excel VBA
- Calculate code metrics per defined entities (files\functions)
- Carry out calculation for 4 releases of both products
- Identify drivers of high complexity/change through interviews with engineers and the reference group
Identifying risky areas of software code in Agile software development

- Correlate metrics to explore their relations and determine which metrics should be selected
- Develop a method by using the selected metrics for identifying and assessing the risks, and establish decision criteria for determining the risk exposure per file
- Identify the risky files and assess the risk using the method and decision criteria
- Collect post-release error reports (ER) per file for 4 past service releases
- Evaluate the method by:
  - Correlating the ERs and calculated risk exposure of files
  - Assigning files to responsible designers for manual assessment (6 weeks period)
- Develop a measurement system according to ISO 15939 to manage the risky files [90]

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 observed 9 metrics of code described in Table 7. These metrics measure such properties of code as size, complexity, dependencies and change frequency. The choice of properties was motivated by:

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

Table 7 presents the metrics 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.

We defined and measured M and NCLOC for files so we could correlate these metrics with ND and NR, and understand their relation. Correlation analysis was carried out as a necessary step for determining which metric of code to choose for risk prediction. Collinear metrics most likely indicate the same property of code. It is important to notice that correlation analyses were not sufficient for selecting metrics so further analysis was also carried out to understand other aspects of metrics’ 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 metrics, Effective_M and Effective_M% presented in Table 7.

Table 7 Metrics and their definitions

<table>
<thead>
<tr>
<th>Name of metric</th>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of non-commented lines of code</td>
<td>NCLOC</td>
<td>The lines of non-blank, non-comment source code in a function/file (this property is measured for both units)</td>
</tr>
<tr>
<td>McCabe’s cyclomatic complexity of a file</td>
<td>File_M</td>
<td>The sum of all functions’ M in a file</td>
</tr>
<tr>
<td>Structural fan out of a function</td>
<td>Sfout</td>
<td>The number of function calls found in a specified function</td>
</tr>
<tr>
<td>Max block depth</td>
<td>MBD</td>
<td>The depth of max nested block in a function</td>
</tr>
<tr>
<td>Number of revisions of a file</td>
<td>NR</td>
<td>The number of check-ins of a file in a specified ClearCase branch and all its sub-branches in a specified time interval</td>
</tr>
<tr>
<td>Number of designers of a file</td>
<td>ND</td>
<td>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</td>
</tr>
<tr>
<td>Effective complexity of a file</td>
<td>Effective_M</td>
<td>The complexity sum of all functions with M &gt; 15 in a file</td>
</tr>
<tr>
<td>Effective cyclomatic complexity percentage of a file</td>
<td>Effective_M%</td>
<td>The ratio of Effective_M and File_M This measure shows how much of the complexity of a file composed by complex function</td>
</tr>
</tbody>
</table>

The aim of the Effective_M% is to show what portion of the File_M number is distributed in complex functions of a file. We calculate them the following way:

\[
\text{Effective}_M\% = \left( \frac{\text{Effective}_M}{\text{File}_M} \right) \times 100 \quad (\text{Eq. 2})
\]
The functions having $M > 15$ are considered complex. In his paper McCabe [91] 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 Effective_M% is shown in Table 8. As the table illustrates, for the specified file the functions 1, 3 and 5 are complex as they have $M > 15$ complexity.

Table 8 An example of calculating Effective_M% metric

<table>
<thead>
<tr>
<th>File name</th>
<th>Function name</th>
<th>$M$</th>
<th>File_M</th>
<th>Effective_M</th>
<th>Effective_M%</th>
</tr>
</thead>
<tbody>
<tr>
<td>file.c</td>
<td>function1()</td>
<td>21</td>
<td>82</td>
<td>58</td>
<td>71%</td>
</tr>
<tr>
<td></td>
<td>function2()</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>function3()</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>function4()</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>function5()</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>function6()</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>function7()</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>function8()</td>
<td>8</td>
<td>8 + 0 + 5 + 17 + 3 + 20 + 8 + 21 = 58</td>
<td>17 + 20 + 21 = 58</td>
<td>(58 / 82) * 100% = 71%</td>
</tr>
</tbody>
</table>

Thus the Effective_M 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 Effective_M% = 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 of a great 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 overall complexity 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. Effective_M% holds an ability to show that file A is complex irrespective its size because it contains complex functions - Effective_M% = 100%.

4 RESULTS

In this section we present the correlation analysis of metrics, designers’ comments on correlations of metrics, the created method for risk assessment and established thresholds.
4.1 Correlation Analysis

Correlation analyses were used to support in determining the choice of the metrics. Table 9 shows the correlations of 4 measures for both products.

**Table 9 Correlation matrix of file measures for both software**

<table>
<thead>
<tr>
<th>Ericsson / Volvo</th>
<th>NCLOC</th>
<th>File_M</th>
<th>ND</th>
</tr>
</thead>
<tbody>
<tr>
<td>File_M</td>
<td>0.91 / 0.90</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ND</td>
<td>0.47 / 0.38</td>
<td>0.41 / 0.40</td>
<td></td>
</tr>
<tr>
<td>NR</td>
<td>0.55 / 0.61</td>
<td>0.48 / 0.68</td>
<td>0.92 / 0.46</td>
</tr>
</tbody>
</table>

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 fundamentally different two reasons behind complex files that are changed often and not complex files that are changed often:

- Not complex files 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
- 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 26 the data points in the scatter plots are the files for telecom software.
Figure 27 shows that strongly correlated (M, NCLOC) and (NR, ND) pairs are crowded along lines with few outliers.

Overall the correlations of measures are similar for both products, yet with an essential difference: Correlation between NR and ND is weak for ECU software. The main reason is that the number of designers of ECU development team is much less than it is for 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 27 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 metrics defined in Table 7 for both products. Table 10 presents correlations of function measures for both products. 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 [57, 92].

<table>
<thead>
<tr>
<th>Ericsson/Volvo</th>
<th>M</th>
<th>NCLOC</th>
<th>MBD</th>
</tr>
</thead>
<tbody>
<tr>
<td>NCLOC</td>
<td>0.75 / 0.77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MBD</td>
<td>0.41 / 0.60</td>
<td>0.24 / 0.44</td>
<td></td>
</tr>
<tr>
<td>Sfout</td>
<td>0.75 / 0.17</td>
<td>0.87 / 0.77</td>
<td>0.34 / 0.12</td>
</tr>
</tbody>
</table>

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, nonetheless there are a few valuable observations worth to notice. Firstly, high cyclomatic complexity necessitates large functions. This implies certain positive correlation between these two metrics. However, big functions are not necessarily complex. This simple observation is well expressed in correlograms of Figure 28 and Figure 29: Scatter plots of (M, NCLOC)

![Correlogram of function measures for telecom software](image1)

![Correlogram of function measures for ECU software](image2)

show that the crowded data have a triangular shape.

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 complexity as a wider concept and cyclomatic complexity number is not observed to be 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. \textit{This means that functions with high M number necessarily have high Sfout.} However the causality of
these two measures is more complicated. For automotive software M and Sfout have rather complicated relationship. In the Figure 29 the scatterplot of (M, Sfout) shows that several functions have high M number and low Sfout. Those are state machines. A few others have high Sfout number 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 measure 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). The most interesting fact about these two metrics is that functions having deepest nested blocks are not the most complex ones.

### 4.2 Selecting Metrics

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

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

As a change metric we selected NR. Firstly several studies has shown that high NR number is an indication of defect prone and difficult-to-maintain code [93, 94]. 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 metric we selected the cyclomatic complexity. Firstly this is a metric 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 at telecom product high cyclomatic complexity entails high Sfout number of functions (Figure 28, 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.
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 metrics 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 metrics 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 metric. 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 Effective_M% measure (Table 7) 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 [93, 94], 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 Effective_M% is low (0.10 / 0.09) motivates us to count the risk as the product of effective complexity and number of revisions.

\[ \text{Relative Risk} = \text{Effective M\%} \times \text{NR} \quad (Eq. 3) \]

This number indicates the likeliness of a file being fault-prone, difficult-to-maintain or difficult-to-manage. We call this kind of combination superposition of metrics as it reflects the joint magnitude of two metrics. 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 Effective_M% 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 Effective_M 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:

- If the Relative_Risk > 1000 for a file in a week development time period then the file is considered highly risky
- If the Relative_Risk > 500 for a file in a week development time period then the file is considered moderately risky.

The advantages of combining the metrics by calculating their product is that it assess the risk irrespective the size of the file: It is only the complexity that matters.

Figure 30 shows how the Effective_M% * NR = 1000 and Effective_M% * 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 as follows:

- Firstly the Relative_Risk of all files is calculated
- Secondly the files are sorted according to Relative_Risk measure, so the first file has the greatest Relative_Risk number, second file has the second greatest Relative_Risk number etc.
- Finally we check files one by one with designers and determine a number that by designers’ perception is a point above which files can be considered risky.
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 stricter threshold 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 the Relative_Risk > 200 for a file in a week development time period then the file is considered risky.

5 EVALUATION

We carried out two steps of evaluation process:

- To evaluate the risk of fault-proneness we correlated number of Error Reports and Relative_Risk of files. This activity enabled to understand if the files having highest Relative_Risk number also have the highest number of errors.
- 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\(^1\). The analysis showed that there is a significant correlation between the two metrics ≈ 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 on them. After

\(^1\) The exact time interval cannot be given for confidentiality reasons
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 [73, 90]. The measurement system runs on daily basis and identifies the files that are risky for current week development time period. Figure 31 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.

![Figure 31 A snapshot of the information product visualizing the risky files](image)

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. They know the most about the details of developed code. The evaluation results at Ericsson were as following:
• 95% of the identified files were confirmed by the designers to be indeed risky
• The rest of the files were reported to have low risk
• Several designers were checking if the file has ER before and deciding risk on that account
• 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
• 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 constant. 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 [47] 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 Effective_M% 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 metrics. Selim, et al. [95] construct survival models based on size metrics 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. [96] use Cox modelling [97] to determine the relative risk of software modules’ defect proneness. Gondra [98] concludes that Machine learning techniques sometimes are not applicable because of scarcity of the data. Pendharkar [99] 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. [100] 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. [101] 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 analysability, changeability, testability and stability. Instead of combining the metrics in one indicator they use experts’ ratings for
internal metrics 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 metrics (for example code clones in some cases have positive effect on quality) and why collinearity of metrics 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. [102] also construct an approach for code analysis and quality consulting but again using various size metrics. 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 metrics useless for assessing riskiness hence we had to construct a new approach which calculates the risk disregarding the size, based on Effective_M% metric. Shihab, et al. [72] present a large-scale industrial study concerned with how code changes can trigger various risks. They use variety of change metrics 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 a wider concept than merely fault-proneness of the code. Secondly, change metrics 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 metrics in one number in order to develop very simple measurement system. Their study observes the most influential metrics 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 $\text{Relative Risk}$ 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 metrics the superposition of two metrics, effective cyclomatic complexity percentage and the number of revisions of a file is a good estimator of risk. The risk is calculated so:

$$\text{Relative Risk} = \text{Effective M\%} \times \text{NR}$$

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 metrics 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 metric 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 metric 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.
PART 5: A METHOD FOR EFFECTIVE REDUCTION OF CODE COMPLEXITY
Abstract
The complexity management of software code has become one of the major problems in software development industry. With growing complexity the maintenance effort of software code increases. Moreover, multiple aspects of complexity create difficulties for optimal complexity management. The objective of this study is to investigate the relationships of several aspects of code complexity and assess the possibilities of optimizing overall code complexity. We conducted an action research project in two software development companies and complemented it with a study of three open source products. Four complexity metrics were measured, and their nature and mutual influence were investigated using statistical methods. The results and possible explanations were discussed with software engineers in industry. The results showed that there are two distinguishable aspects of complexity of source code functions: Those are internal and external complexities which have an inverse relationship. Moreover, the product of them cannot be greater than a certain limit, regardless of software size. We developed a method that permits identification of most complex functions considering the two aspects of complexities. The evaluation showed that the use of the method is effective in industry: It enables identification of few (0.5%) most complex functions out of thousands of functions for manual assessment and reengineering.
A method for effective reduction of code complexity

1 INTRODUCTION

The effect of complexity on software maintenance and fault-proneness has been studied extensively in recent years. There is sufficient evidence that the growing complexity has a direct impact on maintainability and fault proneness of developed software. Many studies have observed that growing complexity triggers decreasing maintainability and escalating fault proneness of software [5, 54, 55]. Banker, et al. [103] was one of the first studies that observed a direct positive relationship of software code complexity and maintenance effort. Later, in order to neutralize the effect of size in investigation of complexity-maintainability relationship, relative complexity measures were applied: Researchers showed that in case of similar size more complex code is significantly more hard-to-maintain and fault-prone [104, 105]. For quantification of different aspects of complexity many measures have been developed [106], among which the most used ones are: McCabe’s cyclomatic complexity, Halstead’s metrics, Chidamber’s and Kemerer’s coupling metrics, depth of nesting, structural fan-in and fan-out. In recent years, these metrics have been used extensively to develop models for assessing the maintainability and fault-proneness of software code [58, 107]. Despite the success of those models the software engineers of industry still have difficulties in effectively reducing code complexity. There is one particular question, posed often: When reducing one aspect of code complexity, how it affects other aspects of complexity? Reformulating this question from complexity reduction perspective, we posed the following research question:

How to reduce code complexity considering various aspects of complexity?

In this paper two distinct aspects of complexity of functions are delineated: Internal and outbound complexities, which have an inverse relationship. Furthermore, the more complex the functions become, the more obvious this relation is. We created a method that assesses the complexity of functions and identifies the most complex ones, considering the two aspects of complexity. An evaluation at Ericsson and Volvo Groups Truck Technology (GTT) showed that the method supports software engineers in industry for effective complexity reduction.

The remainder of this paper is divided into seven sections: First we introduce the motivation of this study, next we describe the design of the study, then we proceed with reporting results. In sections five and
six we present the complexity reduction method and possible threats to its validity. Finally we provide an overview of the related work and conclude the paper.

2 MOTIVATION OF THE STUDY

This research was initiated based on a practical issue that software engineers of industry encountered, when using our previously developed method for software reengineering. In our previous research we had developed a method and supporting measurement system that identified fault-prone and hard-to-maintain files in large development products [43]. The tool supported software engineers to focus on very few files out of thousands for refactoring. The method relied on a combination of two metrics, which were cyclomatic complexity and number of revisions of source code files. During the use of measurement system (about seven months) the engineers of software development teams at Ericsson and at Volvo GTT posed a question regarding refactoring of software: Is that enough to reduce the cyclomatic complexity of a source code function in order to consider that that function’s complexity is significantly reduced? Some engineers argued that while decomposing cyclomatic complexity, the dependencies between functions will increase, and a new type of complexity might be introduced to the functions. Other engineers argued that there are important aspects of complexity that are not captured by measuring cyclomatic complexity. Hence, a further investigation is needed for understanding how to reduce the overall complexity of a function.

In order to better understand the complexity of source code function from practical standpoint and answer the research question, we initiated the research presented in this paper. Particularly we were interested in various aspects of functions complexity and the relationship between them. Furthermore, we pursued finding the possibilities of reducing function’s complexity by considering various complexity aspects of it.

3 STUDY DESIGN

Action research was applied when conducting this study. We adhered to action research principles described in [26, 27]: The researchers carried out the action research project side by side with software engineers in industry. Five steps of action research were executed
throughout the research. The results and possible explanations were discussed with collaborating software engineers.

3.1 Industrial Context

Ericsson and Volvo GTT were collaborating companies in this research. The collaborating organization at Ericsson developed a software product for mobile packet core network. It comprised several hundreds of developers and several cross-functional development teams situated in different geographical positions. The organization at Volvo comprised several hundreds of developers who develop tens of Electronic Control Units (ECU) for trucks of different brands. Our collaboration unit developed two specific ECUs.

3.2 Target Software Products

While the research was conducted by the request of two organizations, we also included three open source products in our research, thus making target number of products being five. Different sizes, domains and development strategies of software products permitted to obtain more generalizable results. The brief description of studied products is as follows:

**Telecom software** was developed by Ericsson. It provided an embedded software system for a packet core network. The software consisted of over two million lines of code.

**Automotive software** was developed by Volvo GTT. It provided an embedded software system for two ECUs of a Volvo truck. The software consisted of over 200 000 lines of code.

**Mozilla Firefox** was free open source software developed by Mozilla Corporation. It provided free web browser for Windows, Linux and other operating systems. The software consisted of over three million lines of code.

**LibreOffice** was free open source software developed by The Document Foundation. It provided office suite for word processing, drawings, slideshows etc. The software consisted of 3.5 million lines of code.

**Gimp** was free software developed by a group of volunteers. It provided graphic editor for image manipulations. The software consisted of over 300 000 lines of code.

All software products were developed partly by C and partly by C++ languages. All products were in development for several years.
3.3 Collected Measures

Source code function was selected as a unit of measurement. This unit is chosen by software engineers of industry and researchers by an agreement that the essential building block in source code is the function. Initially six basic metrics were chosen for this study, based on the credibility of how well they were recognized as software complexity measures in literature and in practice [25, 108]. However, we considered including only such metrics that were previously shown not to have strong correlation together. Strongly correlated complexity metrics characterize the same aspect of code complexity, whereas we pursue investigation of various aspects of complexity. Table 11 presents four selected metrics among six metrics of function. The explanations of why metrics were excluded or included in this study are given in the third column of the table. Additionally, size metrics that often accompany complexity metrics in research are also presented in the table.

Table 11 Complexity and Size Metrics of Source Code Function

<table>
<thead>
<tr>
<th>Metric</th>
<th>Included or excluded</th>
<th>Reasoning of inclusion or exclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cyclomatic complexity</td>
<td>Included</td>
<td>Not correlated with included metrics (NC)</td>
</tr>
<tr>
<td>Structural Fan-in</td>
<td>Included</td>
<td>NC</td>
</tr>
<tr>
<td>Structural Fan-out</td>
<td>Included</td>
<td>NC</td>
</tr>
<tr>
<td>Max. block depth</td>
<td>Included</td>
<td>NC</td>
</tr>
<tr>
<td>Halstead metrics</td>
<td>Excluded</td>
<td>Strongly correlated with cyclomatic complexity [109]</td>
</tr>
<tr>
<td>Size metrics (lines of code,</td>
<td>Excluded</td>
<td>Strong correlation with cyclomatic complexity. High cyclomatic</td>
</tr>
<tr>
<td>Non-commented lines of code,</td>
<td></td>
<td>complexity number necessitates bigger size [43, 110]</td>
</tr>
<tr>
<td>Number of statements)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Derivatives of basic measures, e.g. information flow metrics [111] are also not considered in this study in order to simplify the investigation process. Besides the four basic measures, another nine measures were also calculated during this study based on basic measures. The measures and their definitions are presented in Table 12.
### Table 12 Measures and Definitions

<table>
<thead>
<tr>
<th>Name</th>
<th>Abbrev.</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cyclomatic Complexity (basic measure)</td>
<td>M</td>
<td>The number of linearly independent paths in the control flow graph of a specified function, measured by calculating the number of 'if', 'else', 'while', 'for', '</td>
</tr>
<tr>
<td>Structural fan-in (basic measure)</td>
<td>Fin</td>
<td>The number of invocations of a specified function found in entire source code</td>
</tr>
<tr>
<td>Structural fan-out (basic measure)</td>
<td>Fout</td>
<td>The number of invocations of functions found in a specified function</td>
</tr>
<tr>
<td>Maximum block depth (basic measure)</td>
<td>MBD</td>
<td>The maximum level of nesting found in a function</td>
</tr>
<tr>
<td>Internal complexity</td>
<td>C(I)</td>
<td>The sum of M and Fout</td>
</tr>
<tr>
<td>Outbound complexity</td>
<td>C(O)</td>
<td>Identical as Fin: C(O) = Fin</td>
</tr>
<tr>
<td>Delta cyclomatic complexity</td>
<td>ΔM</td>
<td>The increase or decrease of M of a specified function between two releases</td>
</tr>
<tr>
<td>Delta structural fan-in</td>
<td>ΔFin</td>
<td>The increase or decrease of Fin of specified function between two releases</td>
</tr>
<tr>
<td>Delta structural fan-out</td>
<td>ΔFout</td>
<td>The increase or decrease of Fout of a specified function between two releases</td>
</tr>
<tr>
<td>Delta maximum block depth</td>
<td>ΔMBD</td>
<td>The increase or decrease of MBD of functions between two releases</td>
</tr>
<tr>
<td>Delta internal and outbound complexities</td>
<td>ΔC(I) ΔC(O)</td>
<td>The increase or decrease of internal and outbound complexities of functions between two releases</td>
</tr>
<tr>
<td>Trade-off score</td>
<td>TS</td>
<td>The product of C(I) and C(O). The closer the TS is to 0 the smaller the total complexity of a function is</td>
</tr>
</tbody>
</table>

### 3.4 Research Method

The source codes of all five software products were used for our measurements. The measurements at companies were done by acquiring access to source code repositories of organizations. The
source code of open source products were obtained from publicly available repositories. We have done the measurements on two releases of all software products. The time interval between releases was chosen to be approximately two years. The length of the time interval between two releases was chosen considering that a significant development time is required for observing complexity evolution of overall product.

Two reference groups were formed at two companies to support the researchers with their insights and expertise when evaluating newly obtained results [112]. The software engineers of reference group at Ericsson were an operational architect and two design architects/developers. The reference group at Volvo was formed by a design architect, a developer, a tester and a line manager. Both formal meetings which took place on about monthly basis and informal discussions were main means of communications. The communications in two companies were carried out in a similar manner. The following steps were applied to address the research question in the five phases of action research [26]:

**Diagnosing:** Formulate the research question

**Action design:**

1. Discuss the four basic complexity measures with software engineers in industry and determine what aspects of code complexity these measures reveal (use reference group)
2. Define main (two) complexity aspects, internal and outbound complexity, for a source code function
3. Develop correlograms of four basic complexity measures to visually observe their relationship
4. Calculate outbound and internal complexities based on basic complexity measures
5. Suggest a hypothesis that for complex functions these two complexity aspects might be in an inverse relationship
6. Conduct nonlinear regression and a test of significance to determine if there is an inverse relationship between the two aspects of complexity for complex functions
7. Measure the evolution of four complexity metrics in order to observe the simultaneous changes of complexity measures
8. Define a threshold for complexity trade-off score for functions and indicate the possible highest limit of complexity that a function can have
**Action taking:** Provide functions with highest TS number to software engineers of industry to manually assess and refactor the functions if necessary

**Evaluating:** Assess the effectiveness of how good indicator of complexity TS number is (reference group)

**Specifying learning:** Calibrate results and diagnose the next problem (future research)

## 4 RESULTS

Action research principles lead us to obtain results, discuss it with reference group members, and clarify research direction. For this reason the obtained results and our conclusions with reference groups are reported in a chronological order. We believed that it will make this section comprehensive for reader.

### 4.1 Delineating Two Aspects of Complexity

To start with, we discussed the essence of four basic complexity measures with software engineers at Ericsson and Volvo GTT. The main idea was to understand what these measures show and what they mean for software engineers from development perspective. This knowledge was important from semantic standpoint when analyzing the data with statistical operations.

**McCabe’s complexity** is a measure that shows the number of possible execution paths in a function. Big number of M basically means that there are many conditions to be recalled in a function, when maintaining or understanding it by a developer. Additionally big number of M can increase the difficulties of manual tests, which increases the maintenance effort. McCabe defined a limit of ten for M number, which later was extended to even bigger numbers as thresholds. However, there are two major factors that can influence on establishing a threshold for McCabe complexity number: One is that the source code function can be composed by many different independent code blocks, which make it significantly easy to understand what that function does, irrespective the overall number of condition statements in an a function. This was agreed by all engineers of both reference groups unanimously. The second factor is that if there is a need for recalling a number of conditions at ones, the human memory is retrieving it with groups and associations Miller [113]. These means
that the mind is sorting different patterns into meaningful groups and enhancing the effectiveness of recall process. For instance if there are five similar “if” statements and ten similar “case” statements, human memory most likely will sort them into two groups thus remembering two different things instead of 15.

**Structural Fan-out:** Fout indicates the number of function calls in a given function. A called function demands memory and knowledge from a developer to remember what exactly it does. If the number of called functions is bigger the required memory and knowledge of the developer is also bigger for understanding what exactly all called functions do. The previously described effect for M is the same for Fout measure also: Calling five different functions 45 times in a given F function will require for a software developer to recall five groups of items. Of course, if the arguments of functions are different, things will become more difficult but the number of groups will not change. Both M and Fout measures characterize the internal complexity of functions. They demand a developer to understand and memorize the conditions and calls in the F function in order to maintain that function.

**Maximum Block Depth:** According to developers MBD is a stronger indicator of complexity than previous two measures. The reason is that a deeply nested block obliges a developer to remember all pre-written conditions inside the block in order to understand the current statement. Surely, in a large nested block, sometimes there are similar patterns, which make the recall of them easier, but generally it was agreed that deeply nested blocks are one thing that should be avoided, because they are one of main reasons for emerging defects. One problem though, with MBD as a complexity measure, is that it does not characterize the complexity of a whole function because a function can have many blocks, and their different nesting levels cannot be effectively merged into one number. However the MBD is a strong indicator of complexity which we wanted to include in our analysis to understand its relation with other measures.

**Structural Fan-in:** Fin is a measure which shows how many times an F function is called in other parts of code. As opposed to previous three measures, Fin does not indicate how much complexity is condensed in an F function. It shows the extent that the F function makes other parts of code complex: A developer encountering the F function called in other function should remember what exactly the F function does to be able to maintain other functions correctly. The more the F function is called elsewhere in the code, the more F function’s investment in overall code complexity is bigger. High Fin of the F function means that, in order to make changes in that function a developer needs to understand how
several callers are using that function, to be able to determine the effects of changes on the product.

The first three measures indicate the internal complexity of a given F function for designers. This complexity can cause problems only when maintaining the F function. They all indicate how the elements (statements, conditions, called functions) of F function interoperate inside the functions for completing a particular purpose. Therefore we call these three measures internal complexity measures of a function. The forth measure, Fin, represents the investment of F functions complexity in the overall code. Fin does not show how complex a given function is when maintaining it, but it requires understanding of how that function affects other parts of the code. Thus we call Fin as an outbound complexity measure of a function.

4.2 Visualization of Measures’ Dependencies

Subsequently, we developed correlograms of complexity measures for all five software products in order to understand how the four complexity measures relate to each other. Figure 31 presents correlograms of measures for three products. The correlograms of the other two products are similar two these three, therefore they are not included for enhancing the readability for a reader.

In the figure we can see that for all products, the scatterplots of M and Fout have an emphasized tail which shows a number of crowded functions along with vertical (Fout) axis. We have outlined them with elliptic lines. These functions are mostly executing unit tests. In the same scatterplots we can see a few other functions that have significantly bigger M and Fout numbers then other functions. These are outlined with round circles in the figure. These functions are generated state machines. For example in case of Mozilla the outlined five functions are building hierarchical trees, in case of Ericsson a function is reading and translating vast amount of signals, etc. These state machines have relatively high M and Fout numbers for all products but are much
simpler than the complexity measurements indicate: They are composed by many “switch-case” and simple row of “if” statements. Later in our analysis, we have excluded the outlier functions (outlined round and elliptic lines in Figure 31), which have significantly bigger number of M and Fout numbers than the rest of the functions. This step was done because as we observed the M and Fout numbers have similar magnitude and later in mathematical operations we did not need to standardize these measures. The rest of the data for M and Fout is scattered irregularly showing no specific pattern. The scatterplots of MBD with other measures do not have specific pattern either. The most interesting scatterplots are probably the scatterplots of Fin with other complexity measures. As we can see in the figure three bottommost scatterplots of all correlograms have hyperbolic shapes, clearly indicating an inverse relationship between Fin and other measures (in Figure 31 outlined by a rectangular line for all correlograms). Functions having bigger outbound complexity (Fin) tend to have smaller internal complexity (M, Fout, MBD), and functions with bigger internal complexity tend to have smaller outbound complexity. The explanation of this is simple, yet very important: Functions, which have significant internal and outbound complexities at the same time, are hard-to-maintain. The software engineers of reference groups claim that there are two main aspects of difficulty with this kind of functions:

- It takes significant amount of time to understand how the modified complex function can affect on all its callers
- The functions that have significant internal and outbound complexity sometimes can create a chain of changes in the system which dramatically slows down the maintenance activities.

Conventionally we call that there is a trade-off between the outbound and internal complexity measures, because complexities of both types cannot be high at the same time as shown in Figure 31. We quantify this trade-off in the next section.

### 4.3 Complexity Distribution

In order to observe the trade-off between internal and outbound complexities we decided to combine M and Fout in one number to better express the internal complexity of functions. In our previous research [43] we showed that there is not strong correlation between M and Fout which means their combination will provide a better internal complexity
indicator. We decided not to include MBD measure because as explained before it is not a complexity characteristic of a whole function but rather of a block, which might cause inaccuracies in developing a combined measure. Denoting the combined internal complexity measure by \( C(I) \) we calculate it as:

\[
C(I) = M + \text{Fout} \quad \text{Eq. 1}
\]

We consider that the weights of \( M \) and \( \text{Fout} \) are approximately equal therefore simple sum of them is chosen. Conventionally we denote outbound complexity by \( C(O) \) which is the same as \( \text{Fin} \):

\[
C(O) = \text{Fin} \quad \text{Eq. 2}
\]

By initial observation \( C(I) \) and \( C(O) \) had inverse relationship, therefore we test the significance of this relationship by denoting:

\[
TS = C(I) \times C(O) \quad \text{Eq. 3}
\]

where \( TS \) is called a trade-off score. The value of \( TS \) was calculated as product of internal and outbound complexity numbers. Functions having high \( TS \) number are considered complex, because \( TS \) indicates the joint magnitude of three complexity measures (\( M \), \( \text{Fout} \), and \( \text{Fin} \)). According to our hypothesis the internal and outbound complexities have inverse relationship for complex functions. We chose top 300 functions with highest \( TS \) number and developed a fitted line (Figure 32) for all five software.

![Figure 32 Fitted line plot of Internal and Outbound complexities for top 300 functions with highest TS numbers](image)
One of the well-known lines that describes an inverse relationship is the hyperbola, so we simply selected hyperbola to test if the relation between $C(I)$ and $C(O)$ is hyperbola.

The coefficient of determination for data points (functions) fitting hyperbola was $R^2 \in [0.85, 0.98]$ for five software products. We identified that for approximately 5% of all functions with highest TS numbers, the hyperbola is the line that characterize the relationship of Internal and Outbound Complexities. This observation was true for all five products irrespective their size. We observed that $R^2$ dropped sharply with decreasing TS number of functions. Thus the higher the TS was the better hyperbola fitted to the scatterplots.

We concluded that from certain TS level both complexity measures can no longer increase together in a function, otherwise the function becomes too hard-to-maintain. From semantic point of view, a function that is fulfilling multiple tasks cannot be reused many times, because every reuse might require a slight change, which is difficult to adjust in a complex function.

Next we observed the complexity numbers of top functions with highest TS numbers. Two focal observations were made for this study:

1. **0.2% of all functions in each product with highest TS number had approximately equal TS number across all products irrespective product sizes. This means that no matter how large software products are, the top complex functions of them have equal complexity considering internal and outbound complexities.**

2. **The number of functions having high TS numbers was bigger for bigger products.**

We would like to remind the reader that the sizes of investigated products range from 200 000 lines of code to 3.5 million lines of code, and despite this tremendous difference the complexity numbers for top functions with highest TS numbers were approximately the same for every software product. This observation suggests that no matter how much the overall size and complexity of software increase, there is a limit to how complex its functions can become.

The actual limit of TS number that we derived obtained in our study was $TS(\text{max}) \approx 13000$. By approximation the actual complexity numbers in this limit, on the equilibrium of $M$, $F_{\text{out}}$ and $F_{\text{in}}$ was $M = 80$, $F_{\text{out}} = 80$ and $F_{\text{in}} = 80$.

To assure that there is a maximum limit for TS number we did the following: We measured the complexity evolution over two years. Then, we measured deltas of four complexity measures between two years for all five products and calculated deltas of $C(I)$ and $C(O)$ measures for all
functions. The intention was to observe if for older versions of the products, top TS numbers were exceeded in two years of development period. Figure 33 shows the scatterplot of deltas for Internal and Outbound complexities for all functions of LibreOffice. Every dot represents a function. The vertical and horizontal axes are drawn on zero points to show the position of functions with deltas. If a function is in upper right quarter of the scatterplot of ΔC(I) and ΔC(O) it means that these measures had positive delta together.

![Scatterplot of deltas for Internal and Outbound complexities for all functions of LibreOffice.](image)

**Figure 33 Correlogram of Δ complexity measures for LibreOffice**

As it can be seen on the diagram of Figure 33, there are functions in all quarters of the scatterplot (and also along with axes). This means that there were all types of changes between two versions of software products: C(I) and C(O) were growing together, decreasing together, etc.

![Correlogram of Δ complexity measures for Libreoffice – Top 500 functions with highest TS.](image)

**Figure 34 Correlogram of Δ complexity measures for Libreoffice – Top 500 functions with highest TS**
Afterwards we investigated if outbound and internal complexity measures have increased together for the functions that had highest TS numbers in older versions of product. The Figure 34 presents a similar scatterplot of deltas of complexity measures for top 300 functions with highest TS numbers.

The scatterplot clearly shows that there was no simultaneous growth of C(I) and C(O) measures: Upper right quarter of the scatterplot is empty, indicating that no positive delta exists for top complex functions (emphasized by circle on). Most of the changes are decreases of complexity. By this simple visualization it was shown that there is indeed a complexity limit for functions which previously were articulated by engineers. This limit is not exceeded as it makes functions difficult-to-maintain. At this point we concluded that when considering code complexity reduction, the unit of code that complexity is defined for, is very important, as software developers are working rather with specific units of code than the whole code of product itself. It is worth to mention that when product is constantly growing by size, naturally the overall number and dependencies of functions increase constantly, nonetheless, as our results show, the complexity of a specific function in practice cannot increase perpetually. Above statement might imply that no matter how big software is, if the complexity of its units (functions) is kept in a manageable level then in practice the software can grow even bigger. However this view can be challenged if code units of other abstraction level are also considered, such as software modules and components. Keeping the complexity of functions in a manageable level might not be enough for overall complexity management and additional analysis of complexities of different units might be needed.

### 4.4 Complexity Trade-off

For effective complexity reduction we determined a threshold of TS, which was used for identifying hard-to-maintain functions. Figure 35 presents two lines over C(I) and C(O) scatterplot: First bold hyperbola separates 0.5% of all functions with highest TS score. The 0.5% threshold is established by evaluating the list of top complex functions with software engineers of reference groups as well as considering the amount of functions that can be manually assessed and refactored (if needed) in a release time frame.
A method for effective reduction of code complexity

These functions are depicted as round dark circles in the figure. We call the first hyperbola optimal limit of complexity trade-off. The second hyperbola with dashes is called tolerable limit of complexity trade-off. We consider that there is very little likelihood for functions to be beyond the tolerable limit. The functions that are between optimal limit and tolerable limit are called difficult functions. Considering 0.5% of all functions with highest TS number, by approximation, the optimal line was passing approximately through TS=C(I)*C(O)=1000 point. On the equilibrium of M, Fout and Fin these three complexity measures had the following value: M = Fin = Fout = 23.

In order to evaluate if difficult functions are really difficult to maintain we communicated the results with software engineers in industry. The evaluation results are presented in the next section.

4.5 Evaluation and Calibration of Results

The difficult functions at Ericsson and Volvo GTT were evaluated with engineers of reference groups. The evaluation showed that 80% of difficult functions are hard to maintain due to the nature of their two sides of complexity. Difficult functions were not necessarily the most complex functions considering only internal complexity. However, due to large number of other functions which were dependent on them made them hard-to-maintain. 15% of difficult functions were easy to maintain because of two reasons:

Certain functions were framework functions doing standard small operations. Those have low M number, M~5 but the high Fin of these functions caused them end up in the group of difficult functions.
A method for effective reduction of code complexity

A few functions were state machines with high M and Fout numbers and Fin~2 numbers. These functions were not hard-to-maintain also.

Briefly stating, a functions having very small C(I) could not be regarded as difficult function irrespective how big C(O) was, and a function having small C(O) could not be regarded as difficult functions irrespective how big C(I) was. Therefore we excluded functions that have either very small internal complexity (M & Fout < 5) or very small outbound complexity (Fin < 5).

The remaining 5% of difficult functions were functions, which contained a large number of ‘switch’ and ‘case’ statements, and large number of calls of the same function. These functions, despite big number of M or Fout were easy-to-maintain due to their simplicity. This observation indicates that the use of more sophisticated internal complexity measure can enhance the results obtained from Eq. 3.

The investigation showed that the group of difficult functions in open source products has similar patterns: Simple framework functions and state machines with high TS number were identified. We concluded that in order to automatically exclude these few simple functions from analysis we need to apply more sophisticated complexity measures which was not feasible in this study because of time constraints. However, considering that the number of exceptions was few, we concluded that the method is good enough to be used in collaborating organizations.

Generally the results showed that when developing software it is important to have tolerable complexity for developed units which in our case were functions. It is very important to notice that outbound and internal complexities did not have a causal relationship: For example the decrease of internal complexity does not necessarily mean the increase of the outbound complexity. In fact, all functions that are not in the list of difficult functions have low internal and outbound complexities. This indicates that there is always a way of achieving low complexity on a unit (function) level.

5 A METHOD FOR CODE COMPLEXITY REDUCTION

Generalizing the obtained results we delineate a method for reducing code complexity in software development. The necessary steps for applying the method are as follows:
1. Calculate the values of M and Fout as internal complexity, and Fin as outbound complexity for all functions
2. Calculate the internal complexity C(I) value for all functions, by summing M and Fout: C(I) = M + Fout
3. Calculate the trade-off score (TS) by multiplying internal and outbound complexity values for all functions: TS = C(I) * C(O)
4. Sort all functions in respect with the calculated TS number with descending order
5. Select 0.5% of all functions that have highest TS numbers
6. Review the selected functions for refactoring needs.

The aforementioned first five steps were executed during our research for five large software products as well as the sixth step for automotive and telecom products. The results were shown to be greatly similar.

6 THREATS TO VALIDITY

The validity evaluation of this research relies on a study of Baskerville and Wood-Harper [27] who regard the data validity to be a major problem in action research. While data collection and analysis can be considered “safe” in the phases of diagnosing and action design, there are considerable validity concerns in phases of action taking and evaluating. In action design the data is collected by standard tools, and in order to obtain more generalizable results three open source products (cases) are also investigated. The use of rigorous tools and statistical techniques, and uniformity of the results for all five products secure the validity of conclusions that we had in the end of action design phase. However, the interpretive nature of results creates a tangible validity threat, when making conclusions in the phase of action taking. The problem is that a combination of complexity metrics is not a well-known concept for the engineers of reference group, therefore they interpret complexity numbers and their meanings in their own way while evaluating. If we, researchers explain the concepts to them before the evaluation then we intervene the evaluation by our own interpretations. In this study we did not intervene in the “action taking” and “evaluation” phases relying on the fact that all reference group members are familiar to the complexity concepts.
Using a single measure for internal complexity would facilitate the establishment of an absolute TS threshold which should not be exceeded for any function during development. The difficulty of such threshold’s establishment creates a conclusion validity threat. We could not define such a threshold in a simple way due to the fact that $C(I)$ is not a basic measure which can be calculated directly: It is a combination of two measures. Thus, while applying Eq. 3 for a function, we cannot easily check if the TS number of function is within the established threshold. One solution might be using only $M$ or $Fout$ in Eq. 3 as an internal complexity measure. However both $M$ and $Fout$ are revealing narrow aspects of complexity which can compromise the results. We believe that a single sophisticated complexity measure of internal complexity will enable the establishment of a simple threshold easily practicable in industry, therefore we have planned to address this issue in our forthcoming research. For now, we rely on selecting 0.5% of most complex functions instead of defining an absolute threshold.

The nature of trade-off between internal and outbound complexities can be debatable, which is an internal validity threat: If we observe the distribution of $C(I)$ complexity measure separately we can notice that vast majority of functions in all products have small internal complexity numbers. In our case the number of functions that have $M>10$ cyclomatic complexity, is less than 8% of all functions in the five software products, the number of functions that have $M>20$ is less than 2.5%. The similar effect is observed for other complexity measures. This means that the outcome of higher complexity number has a smaller probability. Hence, assuming that $C(I)$ and $C(O)$ have independent relation, the probability of an outcome of a function that has both high $C(I)$ and $C(O)$ is much smaller due to probability multiplication effect. So, the relation of $C(I)$ and $C(O)$ can be expected to be inverse. However, contemplating the nature of observed trade-off, we can pinpoint two important facts:

1. The statement “the outcome of higher complexity number has a smaller probability” is the manifestation of the nature of complexity itself. This means that small number of complex functions is a result of complexity: It is more difficult to produce and manage complex functions therefore they are more rare in a product.

2. Since there is very small, yet existing probability of a function with both high $C(I)$ and $C(O)$ numbers, there must be a few outlier functions in at least one of the five software products, however there is no such outlier.
Concluding the above discussions we must notice that the **tolerable limit** for all five products are the same, and this supports the claim that no matter what product is developed, what the overall size of the product is, or who the developers are: There is a limit in how far the complexity of functions can grow. Notice that this last statement does not contradict to Lehman’s law of continuous growth of program complexity [114], since Lehman’s law concerns with whole program’s complexity in terms of that program’s deteriorating structure. In this work we emphasize the functions (as units of program) that a developer directly works with.

### 7 RELATED WORK

Probably one of the first studies that reflects on two aspects of code complexity, internal and outbound complexity, is Henry’s and Kafura’s study where they define complexity metrics based on information flow between software components Henry and Kafura [111]. In this paper the authors define the complexity based on the product of Fan-out and Fan-in. In another paper of them, they investigate the correlation of three widely known complexity metrics to understand their relationship [109]. We believe that their research had a pivotal role in exploring complexity management issues and we continued the tradition in term of scrutinizing the nature of various aspects of complexity.

A comprehensive list of software complexity types can be found in Mens [115] work but the discussion of nature and relations of complexity types was out of the scope of that study. Card and Agresti [116] distinguish local and inter-module complexity for software code and define the sum of them as a total complexity. Nonetheless, they did not investigate the relationship of the two complexity aspects, which was a topic for our study. Concas, et al. [117] are studying the use of new complexity metrics, which are based on social network analysis. They correlate them with traditional complexity metrics and investigate their dependencies. In their work Councell and Mubarak investigated the relationship of Fan-in and Fan-out of classes and found that there are ‘server’ and ‘client’ types of classes which just have high fan-in and high fan-out [118, 119]. What is more, they found classes which have relatively high fan-in and fan-out but they never investigated how much these two measures can be scaled up together. Our study is a relevant complement to their study as it investigates further the trade-off between complexity measures and defines two types of complexity aspects that have an inverse relationship for complex functions.
A method for effective reduction of code complexity

Tran-Cao, et al. [120] define the code complexity as a three-dimensional vector, which is composed by data movement, data manipulation and system complexity. They show that effort estimation models based on combination of these complexity metrics is more effective than previously suggested models. Xiao [121] proposed a hybrid metric for internal complexity of functions, which is based on Halstead’s, McCabe’s complexity metrics and fan-out. We have planned in our future work to use more sophisticated metric for measuring the internal complexity, as it can reveal wider aspects of internal complexity of functions.

8 CONCLUSIONS

The purpose of our research was to investigate the relationship of various aspects of software complexity for effective complexity reduction. We analyzed two industrial and three open source products for this purpose. The results showed that there are two main types of complexity that emerge when developing software functions, internal complexity and outbound complexity. The former is complexity related to internal operations of functions which makes the function difficult to understand. The latter shows the ability and extent that a given function can make other parts of code complex.

Analysis showed that:
- Outbound and internal complexities have an inverse relationship
- Irrespective the size and domain of software there is a clear maximum limit for the product of outbound and internal complexity measures

These two investigations outlined the existing trade-off between two complexities, that is, in practice the product of two complexity values for a function cannot exceed a certain limit.

The paper also presents a method for effective complexity reduction. The evaluation showed that the method enables to identify the few most complex functions out of thousands of functions for manual assessment and reengineering.

Acknowledgment

We express our thanks to the companies for their support in the study. This research was carried out in the Software Centre with collaboration of Chalmers, University of Gothenburg, Ericsson, and Volvo Group Truck Technology.
PART 6: PROFILING PRE-RELEASE SOFTWARE PRODUCT PERFORMANCE
Abstract

Large software development organizations require effective means of quantifying excellence of products and improvement areas. A good quantification of excellence supports organizations in retaining market leadership. In addition, a good quantification of improvement areas is needed to continuously increase performance of products and processes. In this paper we present a method for developing product and organizational performance profiles. The profiles are a means of quantifying pre-release properties of products and quantifying performance of software development processes. We conducted two case studies at three companies – Ericsson, Volvo Group Truck Technology and Volvo Car Corporation. The goal of first case study is to identify risky areas of source code. We used a focus group to elicit and evaluate measures and indicators at Ericsson. Volvo Group Truck Technology was used to validate our profiling method. The results of the first case study showed that profiling of product performance can be done by identifying risky areas of source code using combination of two measures – McCabe complexity and number of revisions of files. The results of second case study show that profiling change frequencies of models can help developers identify implicit architectural dependencies. We conclude that profiling is an effective tool for supporting improvements of product and organizational performance. The key for creating useful profiles is the close collaboration between research and development organizations.
1 INTRODUCTION

Continuous assessment of product and development performance is a means to support developers in visualization of product status and proactive decision making. In modern software development organizations this assessment process is usually a complex activity. Glass [122] observed that for every 25% increase in problem complexity there is 100% complexity increase in software solution. The assessment of contemporary software products is difficult because there are no explicit properties of software which can be directly used to quantify excellence, as unlike other products software products are intangible and require visualization.

The focus of our research project in Software Center is to identify and develop methods and tools for profiling excellence of software products together with the collaborating companies. The goal of this paper is to present a process for profiling pre-release software product performance. This goal has been accomplished by conducting action research at Ericsson, Volvo Cars Corporation, Volvo Group Truck Technology (GTT) and Saab Defense. By collaborating closely with industrial partners we developed different profiling tools and evaluating them for industrial use. The results show that close collaboration with developers facilitates creating useful profiling tools. Two profiling tools that are developed in companies are presented in this paper.

The rest of the paper is organized as follows: Firstly, we introduce the concept of profiling in the context of large software development organizations, next we present two case studies about pre-release product and process profiling. Finally we describe the experiences and learning of companies and draw conclusions from this paper.

2 PROFILING

To make quick and optimal decisions, managers, architects, designers and developers need to have good insights on how the developed product or development processes are implemented and how they evolve over time. Mere subjective estimates and opinions might be good enough for small application’s development but for large complex products quantitative measures are required. Example of measures for profiling could be the ratio of executed tasks and planned tasks, system testing queue over time, throughput trend, architectural dependencies between components, complexity of code etc.
Generally “assessment” and “evaluation” are widely used by researchers when it comes to discussing product excellence – for example when conducting benchmarking [123, 124]. However, these concepts are focused on the evaluation of software products. The terms “assessment” or “evaluation” are intended to determine the degree of excellence of a particular entity: Examples of entity can be a single source code function, a model, architecture of product, etc.

The term profiling opens a different perspective when considering about product excellence. It is a neutral concept and regards an entity as a composition of elements. Such examples can be software code composed by source code functions, software architecture composed by architectural components, etc. When profiling the excellence of code instead of merely assessing how good the code is, the functions are assessed and presented in one picture. In Figure 36 we can see that the source code can be assessed to be good (left-hand picture) or profiled in terms of a defined criterion for the excellence of functions (right hand picture). For example if the criterion is the maintainability of functions then three levels of maintainability may be defined and represented by colors. In Figure 36 the darkest quadrats represent functions that are hard to maintain. Moderately dark quadrats represent functions with moderate maintainability and light grey represents functions that are easy to maintain (right-hand picture).

![Assessing source code vs Profiling source code](image)

**Figure 36 Difference of assessment and profiling of source code**

Generalizing we can state that profiling the excellence of an entity is a representation of elements comprising that entity, where the assessments of all elements are available. Although each function can be assessed separately, the essence of profiling assumes that the whole source code, its functions and their assessments are represented just in one picture. This property is opening a new dimension for evaluation, which is the comparison of functions in one picture or comparison of two products' source code with equivalent representation. In this paper we define profiling as:
Definition 1: Profiling pre-release software product performance is the measurement and representation of such properties of product, which enables evaluation of that product’s excellence before delivery.

This definition emphasizes profiling related to pre-release product properties (e.g. internal quality attributes such as complexity) and post-release success of the product (excellence). We define profiling organizational performance in similar terms:

Definition 2: Profiling software organizational performance is the measurement and representation of such properties of organizational processes that enables evaluation of organizational performance in product development.

According to definitions profiling must enable evaluation of excellence. An example of profiling is measuring size, complexity, trend of reported defects and visualizing them for designers so that problem areas are easily visible.

3 PROFILING PRE-RELEASE PRODUCT AND PROCESS PERFORMANCE IN LARGE SOFTWARE DEVELOPMENT ORGANIZATIONS

The software development companies, which we collaborate with, use Agile principles to assure quick response on customers’ requirements. This kind of development usually comprises diverse activities with complex tasks as the product itself is big and complex.

Developing these products is characterized by such challenges as mixed long-term planning for main release, short time planning for service releases, distributed decision making by software development teams, communication between teams, or multisite development. Figure 37 presents an overview of Agile development in collaborating companies.

![Figure 37 Feature development by Agile methodology](image)
The functional requirements (FR) and non-functional requirements (NFR) are prioritized and packaged into work packages by product management (PM), then systemized by system management (SM) and at last implemented and tested by design management (DM) and test teams. Each team delivers their code into the main branch.

Before the release the development teams are concerned with how good the developed artifacts are and how well the development processes are carried out. Pre-release product profiling is concerned with representing the excellence of developed artifacts and development processes. Improvements of artifacts and processes have a twofold effect: Decreasing internal development costs and efforts, and implicitly ensuring better quality of released software. The ultimate goal of development teams is to release a product that is complete by functionality and fulfills all the requirements of quality (reliability, usability, efficiency etc.). Therefore, pre-release improvements of software artifacts and well-designed processes create a high likelihood for a high quality product.

4 ESTABLISHING THE PROCESS OF PROFILING

In this section we define the process of profiling and provide examples. Before starting the profiling process it is important to consider what exactly should be profiled and what should be achieved by that – i.e. elicit the information need for the profile [125]. In order to develop a profile of the excellence of pre-release software the profiling process can be designed as follows:

1. **Identify measurable properties of the product artifacts that allow the assessment of the product’s excellence:** Well-known artifacts are software requirements specification, architecture, components, source code files, functions etc. Well-known properties are ambiguity of requirements, architectural dependencies, code complexity etc.

2. **Identify measurement tool and measure the specified properties of the artifact:** Example measures for properties are lines of code (LOC) as size measure, McCabe complexity number as complexity measure, structural fan-out as dependency measure, ambiguity ratings in a scale of one to five for requirements etc.

3. **Identify the influence of measure on artifact’s excellence:** Analytically or empirically established thresholds for measures by
which the developers can assess the excellence of the artifact. For example a threshold for fan-out as a dependency measure could be seven. If a component has more than seven fan-out then it is considered too vulnerable to external changes.

4. **Reduce/optimize the number of measured properties by using statistical methods:** Several properties can be related to each other or metrics might show weak influence on product excellence. For example Number of statements and Number of lines of code are showing the same property of code, size, thus one measure should be used. Another example is the number of functions in a file have weak influence on maintainability and fault proneness of code so it cannot be used to assess maintainability of code.

5. **If possible combine the remaining independent measures using statistical or mathematical methods in a way that they jointly characterize the excellence of the artifact:** For example requirements ambiguity and complexity ratings can be mixed to jointly represent the difficulty that developers have in understanding the requirements.

6. **Define thresholds for the joint measure to separate a number of artifacts by their excellence:** For example if a function has greater than 20 cyclomatic number and is called by other functions more than 50 times than it is considered to be a badly designed function.

7. **Represent the artifact as it is composed by elements, where elements are assessed by combined the measure and thresholds:** For example the source code (artifact) of product with its files (elements) can be represented by measured sizes (characteristic) of files. Files having more than 1000 (threshold) lines of code (measure) are considered big (assessment) the rest are considered normal (assessment).

Before implementing these steps there are two essential considerations about the quality of measures:

- **Accuracy:** How accurately the measure counts the property of an artifact?
- **Appropriateness:** Does the measure support in decision making?

In our previous work [126] we showed that effective use of measures, as support for decision processes, requires about 20 measures at the top management level.
The risk of developing wrong profile for an artifact is high as the developers not always know what selected measures actually show and how profound basis the established thresholds have. That is why the researchers in the field of software engineering should work beside engineers in order to guarantee that measurement methods and tools have scientific basis.

In the next two sections we are giving an overview of how two example measures are developed and selected at large software development companies. One measure is designed for profiling pre-release product performance. The second one is designed to profile the development process.

5 CASE STUDY: DEVELOPING RISK PROFILE OF SOFTWARE CODE

An action research project was conducted at two companies, Ericsson AB and Volvo GTT with the aim to create a profile that visualizes the risk of source files. In this context we define the code to be risky if it is fault-prone, potentially difficult-to-maintain or difficult-to-manage. As result of conducted project we developed a method and tool that locates risky source code files. An overview on how the method was created and its usage is presented beneath.

The profiling process starts with identification of code properties that should be measured. According to Definition 1 these properties must enable assessment and representation of product excellence. In our case, a specific measure of excellence is the riskiness of the code. The more risk-free the better excellence the code has. Identifications of these properties should have either scientific or intuitive basis. Thus we measured properties of code that:

- are manifested in literature to have influence on fault-proneness or maintainability of code
- are confirmed by many experienced developers to have correlation with their difficulty of managing code

The measured properties are divided into categories and presented in Figure 38. The Δ letter in the figure indicates the change of the metric over a specified time interval.
We followed action research principles for measurement and selection of metrics. Close collaboration with a reference group of designers at Ericsson, which was administered by the line manager of developed product, allowed us to discuss intermediate results. During the period of 8 months we conducted analyses and met biweekly with reference group to get feedback on presented results. Figure 39 illustrates the applied research method that is compliant to the process described in section 4.

An Overview of profiling process carried out at Ericsson

The more detailed information of what research activities have been carried out, what results were obtained and presented in each step, and the feedback of designers are presented in Table 13.
Profiling pre-release software product performance

Table 13 Biweekly analyses of results and reference group’s feedback at Ericsson

<table>
<thead>
<tr>
<th>Week</th>
<th>Presented</th>
<th>Feedback</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Size, complexity and their evolution between two main releases of the product is observed and presented</td>
<td>Skip measuring size. Measure the evolution of complexity between 4 main releases to find out if the complexity increases constantly. Observe complexity difference between C and C++ code.</td>
</tr>
<tr>
<td></td>
<td>Strong correlation of size and complexity is found</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>The complexity of the product increases constantly through 4 releases. C code generally contains more complex functions than C++ code.</td>
<td>The overall complexity of the product should increase because it is inherent to increasing functionality. Instead it is important to measure if the number of complex functions is increasing.</td>
</tr>
<tr>
<td>5</td>
<td>The number of complex functions is increasing over development time.</td>
<td>Investigate top functions with highest complexity increase. Find patterns associated with complexity increase and trigger action if necessary.</td>
</tr>
<tr>
<td>7</td>
<td>The causes of top 30 functions with complexity increase are investigated.</td>
<td>The causes are discussed. The reference group decides upon how change report policy should be.</td>
</tr>
<tr>
<td>9</td>
<td>Similar measurements are carried out on code generated from models</td>
<td>Develop an initial measurement system for tracking complexity increases. Measure other proposed properties that might have influence on fault-proneness and maintainability</td>
</tr>
<tr>
<td>11</td>
<td>Number of revisions of source files is measured. Clustering technique is introduced.</td>
<td>Thus measure the number of designers as they might have stronger influence</td>
</tr>
<tr>
<td>13</td>
<td>Number of designers is measured. Strong correlation is found with number of revisions</td>
<td>Skip measuring number of designers</td>
</tr>
<tr>
<td>15</td>
<td>Correlation of revisions and error reports is moderate positive: Not all files with high revisions are problem</td>
<td>Combine available independent metrics to obtain stronger indicator of problematic files</td>
</tr>
<tr>
<td>17</td>
<td>fan-out, block depth are measured strong correlation between complexity and fan-out is presented</td>
<td>No particular feedback. Fan-out and block depth are skipped</td>
</tr>
<tr>
<td>19</td>
<td>Different clustering techniques are applied to combine complexity and revisions. None of them is suitable.</td>
<td>No particular feedback. Waiting for successful combination of metrics</td>
</tr>
<tr>
<td>21</td>
<td>The product of complexity and revisions is proposed as combined metric. Two thresholds are defined for</td>
<td>The method is intuitive and should be evaluated. The reference group decides to conduct 6 weeks</td>
</tr>
</tbody>
</table>
distinguishing low, moderate, and high risk containing files | evaluation period
---|---
23-27 Evaluation and replication of analysis at Volvo Trucks | Evaluation shows that the proposed method is a good metric for giving risk profile of source code files

The table illustrates the short feedback loops and changes in the focus of the project/profile over time – aligned with the model presented by Sandberg et al [127]. While the discussions with designers allowed us to select metrics and understand reasons why certain files can be very complex or much changed, the statistical techniques were used to limit the number of measures. It is common that when having many different measures some of them might be strongly correlated and, in fact, show the same property of code. For example if we measure the number of statements and the number of lines of source code files, we can find that they are strongly correlated. The reason is they both measure the same property of code – size. With pairwise correlation analyses and discussing the relations of measures with designers, we could reduce the number of measures to two independent ones, thus avoiding redundant measures. One of the examples of correlation is illustrated in Figure 40.

In the figure every dot represents a c function of telecom software.

![Figure 40 Correlation of complexity and size](attachment:image)

The scatterplot of functions shows that not all big functions are complex: There are many functions along with “Lines of code” axis showing zero complexity and many functions are in the left uppermost corner of graph showing big size and small complexity. Conversely, there
is no function in down most side showing that there is no complex function with small size. This means that if we chose size as a measure for risk identification we might omit the complexity aspect of code, as not all big functions are complex, whereas choosing complexity as a measure, we know that size is also involved in this measure, as complex functions are also big.

Doing similar analysis with all metrics and discussing results with designers we could reduce the number of measures to 2 independent ones – McCabes cyclomatic complexity and Number of revisions of files. We combined these two metrics to obtain their joint magnitude as an indication of risk. Denoting the risk as the product of complexity and number of revisions of files we get:

\[ \text{Risk} = \text{Complexity} \times \text{Revisions} \]

This formula permits to assess the riskiness of files. Then two thresholds were established for risk level. For example the files that have Risk > 400 score are considered very risky, 200 < Risk < 400 are moderately risky etc. One simple example of grouping files according to their risk level is shown in Figure 41.

![Figure 41 Risk profile of source code files](image)

The figure presents the risk profile of source code as a composite of files. All the files of product are divided into three groups:

- The round dots on the scatter are files with high risk
- The squares are files with moderate risk
- The triangles are files with no or little risk
The established threshold can vary from company to company, depending on how much complexity developers can tolerate, what the number of revisions show, how many risky files the developers can manage to refactor or test. The thresholds vary but, as evaluation showed, the fact that files having many revisions and high complexity are risky is not likely to change.

After assessment we evaluated the method for 6 weeks period with designers. The method was confirmed to be accurate and relevant for risk assessment. Both designers’ feedback and correlation between risk and error reports show that the method is viable in systematic industrial use for risk assessment. After evaluation the measurement system was developed to give continuous feedback to designers and management on risky files (Figure 42).

During the research we measured 12 properties of code, identified relevant properties to select, discussed the intermediate results with experts to assure that the research is going to right direction, created the method and evaluated it.

6 CASE STUDY: PROFILING CHANGE FREQUENCY OF MODELS OVER TIME

In this section we discuss a case study of a specific process profiling carried out at Volvo Cars Corporation. The development team of one of the electronic control units at Volvo was concerned with how to profile the changes of product development models over time. The motivation was that if it is possible to visualize how frequently models change over time with respect to continuous development and maintenance, designers can draw conclusions on which models the most development
efforts are focused on, and if development of one model triggers changes in other models. This information can help designers to understand if they consume their development efforts as it is distributed in their time-plan and if the dependencies between models are compliant with the designed architecture. In this case the profiling process described in section 4 is much simpler as there is only one property, that should be measured – number of changes.

Figure 43 visualizes the change frequency of simulink models over development weeks described in Feldt, et al. [128]. Every line in the figure corresponds to a model, whilst columns are development weeks.

The darker spots in the figure are models with more frequent changes within the same week. This kind of profile of changes enables developers to focus on most frequent changes and explore the reasons of them. Several reasons can be behind frequent changes: These can be new functionality development, error corrections or complex models requiring relatively much time for maintenance. Depending on the reason of changes the actions are different: such as “no action required”, “redesigning unwanted architectural dependencies” etc. Another benefit of this specific representation is that change patterns between models
can be identified. For example one can observe that every time changing model A after two weeks model B is changed. Intrinsic dependencies that might be among A and B can be identified and managed.

As we see this kind of profiling does not require explicit establishment of thresholds but it does not mean that the thresholds do not exist and no action is required. The figure visualizes models’ change frequency which means that there are always a few models that are changed most frequent (darkest ones in the figure). In practice software designers are aware of what changes are expected and by checking these few models they can make sure if any unwanted changes have occurred. In case of occurrence designers can do architectural conformity check and redesign models if necessary.

Change frequency profiles of models are used at Volvo Cars Corporation for systematically monitoring the compliance of developed models with architecture, and finding hidden dependences between models.

7 RELATED WORK

Robillard, et al. [129] is one of the early studies that attempts to profile the pre-release software product by means of calculating and visualizing all available metrics at the time. They organize the visualization in a compact and simple way so stakeholders with different backgrounds can easily grasp the info. Kitson and Masters [130] investigate possibilities of profiling software processes and categorizing according to their maturity. While software products are becoming more and more complex over years more sophisticated techniques are required to enable holistic profiling the performance of pre-release product and processes. Today there are numerous studies that are providing methods and tools for profiling different aspects of organizational performance [131-134]. But before introducing how our research is concerned with profiling we need to define what is profiling the performance of pre-release product and processes.

Profiling of pre-release product and organization performance can support to types of decisions: (i) related to the economics of software product development [135] – referred to as managerial in this paper and (ii) decisions related to technology used in product development [136] – referred to as technical in this paper. Ruhe [136] recognizes a wider spectrum of decisions in software engineering – e.g. project planning and control, architectural and design decisions, requirements. However, in the studied organization it was found that it is easier to discuss metrics
and decisions in the chosen two categories without any loss of generalizability, while putting stress onto the interplay between decisions and metrics.

Lawler and Kitchenham [64] provided an approach for aggregating measures across organizations and presenting aggregated measures for managers – which is similar to profiling. Although the approach in itself is similar to ISO/IEC 15939 [137] the studied organizations does not use aggregated measures as they do not provide the possibility to quickly guide improvements in the organizations – and in the extreme cases led to measures and indicators that were hard to interpret and backtrack which events caused the indicators to change status, e.g. [132]. Lawler and Kitchenham’s approach is similar to the approach used in modern business intelligence tools which aim at providing stakeholders with all available information on request. Although this approach is promising and used in mature disciplines, like mechanical engineering, with established metric systems and theoretically well-grounded measures, the approach has high risks in software development organizations. The risks are related to the potential misinterpretation of data across different projects and products (e.g. even the simplest measures like lines of code can be measured in multiple ways).

Organizations starting to use business intelligence tools often face the problem of using these tools in an efficient way after overcoming the initial threshold of establishing the infrastructure for the tools. Elbashir et al. [138] studied the problems of measuring the value that business intelligence tools bring into organizations in a longer run and concluded that these tools are spreading from strategic decision support to support decisions at the operational levels in the company. The value of measures from these tools, according to Elbashir et al, calls for more research. Profiling presented in this paper supports organizations in effective use of business intelligence.

Balanced Score Cards and corporate performance management tools are often considered at top management level as methods and tools for controlling the performance of organization [139-142]. The traces of the Balanced Score Card approach were observed at the top management level in our previous work [143]. The studied organization took these measures one step further – making them precise, operational and automated (in many cases). Profiling helps the organizations in choosing the right measures for each scorecard.

Completeness of information is an important aspect in profiling. It is often a part of the overall information quality and its evaluation. The basis for our research is one of available frameworks for assessing information quality – AIMQ [144]. The framework contains both the
attributes of information quality, methods for measuring it and has been successfully applied in industry in the area of data warehousing. In our research we have taken the method one step further and developed a method for automatic and run-time checking of information quality in a narrowed field: measurement systems [145]. In this work we present a method for assessing how complete the information products are: This is a part of requirements for having high-quality metrics. There exist several alternative (to AIMQ) frameworks for assessing information quality, which we also investigated, for example Kahn et al. [146], Mayer and Willshire [147], Goodhue [148], Serrano et al. [149]. The completeness of information is present in all of them in different forms. The AIMQ framework was chosen as it was previously used in our research on information quality – where the information completeness is a part of.

Burkhard et al. [150] found that although the indicators are presented visually, people are surrounded by overwhelming information and miss the big picture. This “bigger picture” in the context of monitoring of software product development means that the stakeholders need to monitor entities that they formally do not manage. For example project managers monitor projects but also need to understand how the “product has it", for example what the quality of the developed product is. For stakeholders responsible for parts of product development that means that they need to understand what the situation “upstream” is – i.e. whether there are any potential problems that might affect their work after a period of time.

8 CONCLUSIONS

Profiling product and organizational performance is concerned with assessing and representing the whole product and process excellence in one picture, where comprising elements of the product are visible in that picture. The method for developing profiles presented in this paper addresses such issues as what the profile should show, which elements should be profiled as building blocks of product and how the profile of product or process will help in decision making. In this paper we presented a method which addresses these issues. We presented two industrial experience report on how we developed risk profile of product at Ericsson and change frequency profile of models at Volvo Car Corporation. Both reports are relying on the profiling method presented in this paper.
The elements that comprise the product are different depending on product characteristics – e.g. source code functions, architectural components, models, and requirements specification. At Ericsson the elements that comprise the product were source code files which led to one set of elements in the profile whereas at Volvo Car Corporation the main elements were simulink models which resulted in a different set of elements in the profile.

The developed risk-profiles helped designers to detect the most risky few files out of thousands and refactor them. The change frequency profile helped to detect hidden architectural dependencies and redesign models if necessary.

The next steps in our research are to expand the set of available measures to requirements specifications, architecture level metrics and test metrics. The expansion could provide the possibility to include a wider spectrum of stakeholders in the decision-making and analysis of particular profiles.
REFERENCES


References


References


References


References


References


References


References


References


