TKK Dissertations 160
Espoo 2009

# SOFTWARE EVOLVABILITY – EMPIRICALLY DISCOVERED EVOLVABILITY ISSUES AND HUMAN EVALUATIONS

Doctoral Dissertation

**Mika Mäntylä**

**Helsinki University of Technology**
**Faculty of Information and Natural Sciences**
**Department of Computer Science and Engineering**

TKK Dissertations 160
Espoo 2009

# SOFTWARE EVOLVABILITY – EMPIRICALLY DISCOVERED EVOLVABILITY ISSUES AND HUMAN EVALUATIONS

Doctoral Dissertation

**Mika Mäntylä**

Dissertation for the degree of Doctor of Science in Technology to be presented with due permission of the Faculty of Information and Natural Sciences for public examination and debate in Auditorium T2 at Helsinki University of Technology (Espoo, Finland) on the 8th of May, 2009, at 12 noon.

**Helsinki University of Technology**
**Faculty of Information and Natural Sciences**
**Department of Computer Science and Engineering**

**Teknillinen korkeakoulu**
**Informaatio- ja luonnontieteiden tiedekunta**
**Tietotekniikan laitos**

Distribution:
Helsinki University of Technology
Faculty of Information and Natural Sciences
Department of Computer Science and Engineering
P.O. Box 9210 (Tekniikantie 14)
FI - 02015 TKK
FINLAND
URL: http://www.cse.tkk.fi/
Tel.  +358-9-451 4851
Fax  +358-9-451 4958
E-mail: mika.mantyla@tkk.fi

ISBN 978-951-22-9856-3
ISBN 978-951-22-9857-0 (PDF)
ISSN 1795-2239
ISSN 1795-4584 (PDF)
URL: http://lib.tkk.fi/Diss/2009/isbn9789512298570/

TKK-DISS-2592

Multiprint Oy
Espoo 2009

| | |
|---|---|
| ABSTRACT OF DOCTORAL DISSERTATION | HELSINKI UNIVERSITY OF TECHNOLOGY<br>P.O. BOX 1000, FI-02015 TKK<br>http://www.tkk.fi |

Author Mika Mäntylä

Name of the dissertation
Software evolvability – Empirically discovered evolvability issues and human evaluations

| | |
|---|---|
| Manuscript submitted November 11, 2008 | Manuscript revised April 14, 2009 |

Date of the defence May 8, 2009

| | |
|---|---|
| ☐ Monograph | ☒ Article dissertation (summary + original articles) |

| | |
|---|---|
| Faculty | Faculty of Information and Natural Sciences |
| Department | Department of Computer Science and Engineering |
| Field of research | Software Engineering |
| Opponent(s) | Professor Magne Jørgensen |
| Supervisor | Professor Reijo Sulonen |
| Instructor | Professor Reijo Sulonen |

Abstract
Evolution of a software system can take decades and can cost up to several billion Euros. Software evolvability refers to how easily software is understood, modified, adapted, corrected, and developed. It has been estimated that software evolvability can explain 25% to 38% of the costs of software evolution. Prior research has presented software evolvability criteria and quantified the criteria utilizing source code metrics. However, the empirical observations of software evolvability issues and human evaluations of them have largely been ignored.

This dissertation empirically studies human evaluations and observations of software evolvability issues. This work utilizes both qualitative and quantitative research methods. Empirical data was collected from controlled experiments with student subjects, and by observing issues that were discovered in real industrial settings.

This dissertation presents a new classification for software evolvability issues. The information provided by the classification is extended by the detailed analysis of evolvability issues that have been discovered in code reviews and their distributions to different issue types. Furthermore, this work studies human evaluations of software evolvability; more specifically, it focuses on the interrater agreement of the evaluations, the affect of demographics, the evolvability issues that humans find to be most significant, as well as the relationship between human evaluation and source code metrics based evaluations.

The results show that code review that is performed after light functional testing reveals three times as many evolvability issues as functional defects. We also discovered a new evolvability issue called “solution approach,” which indicates a need to rethink the current solution rather than reorganize it. For solution approach issues, we are not aware of any research that presents or discusses such issues in the software engineering domain. We found weak evidence that software evolvability evaluations are more affected by a person’s role in the organization and the relationship (authorship) to the code than by education and work experience. Comparison of code metrics and human evaluations revealed that metrics cannot detect all human found evolvability issues.

Keywords Software evolvability issue, Refactoring, Controlled experiment, Observation, Interrater agreement, Code review

| | |
|---|---|
| ISBN (printed) 978-951-22-9856-3 | ISSN (printed) 1795-2239 |
| ISBN (pdf) 978-951-22-9857-0 | ISSN (pdf) 1795-4584 |
| Language English | Number of pages 66 p. + app. 80 p. |

Publisher Helsinki University of Technology, Department of Computer Science and Engineering

Print distribution Helsinki University of Technology, Department of Computer Science and Engineering

☒ The dissertation can be read at http://lib.tkk.fi/Diss/2009/isbn9789512298570/

| | |
|---|---|
| VÄITÖSKIRJAN TIIVISTELMÄ | TEKNILLINEN KORKEAKOULU<br>PL 1000, 02015 TKK<br>http://www.tkk.fi |

Tekijä Mika Mäntylä

Väitöskirjan nimi
Ohjelmistojen jatkokehitettävyys – Empiirisesti havaittuja jatkokehitettävyysongelmia ja ihmisten arvioita

| | |
|---|---|
| Käsikirjoituksen päivämäärä 11.11.2008 | Korjatun käsikirjoituksen päivämäärä 14.4.2009 |

Väitöstilaisuuden ajankohta 8.5.2009

| | |
|---|---|
| Monografia | [✓ Yhdistelmäväitöskirja (yhteenveto + erillisartikkelit)] |

| | |
|---|---|
| Tiedekunta | Informaatio ja luonnontieteiden tiedekunta |
| Laitos | Tietotekniikan laitos |
| Tutkimusala | Ohjelmistotuotanto |
| Vastaväittäjä(t) | Professori Magne Jørgensen |
| Työn valvoja | Professori Reijo Sulonen |
| Työn ohjaaja | Professori Reijo Sulonen |

Tiivistelmä

Ohjelmistojärjestelmien evoluutio saattaa kestää vuosikymmeniä ja maksaa miljardeja euroja. Ohjelmiston jatkokehitettävyys ilmaisee kuinka helppoa ohjelmiston lähdekoodia on ymmärtää, muokata, korjata ja kehittää edelleen. On esitetty, että ohjelmiston jatkokehitettävyys selittää 25–38% ohjelmistoevoluution kustannuksista. Aiempi tutkimus on lähestynyt ohjelmiston jatkokehitettävyyden arviointia lähinnä jatkokehitettävyyskriteerien ja niistä johdettujen lähdekoodimittareiden avulla jättäen suurelta osin ottamatta huomioon tavallisten ohjelmistokehittäjien kentällä tekemät empiiriset havainnot.

Tässä väitöstyössä tutkitaan empiirisesti ihmisten arvioita ohjelmiston jatkokehitettävyydestä ja ihmisten tekemiä havaintoja ohjelmakoodin jatkokehitettävyysongelmista. Työssä sovelletaan sekä laadullisia että tilastollisia tutkimusmenetelmiä. Empiirinen aineisto on kerätty kontrolloiduilla opiskelijakokeilla ja havainnoimalla teollisuuden todellisissa tilanteissa ilmenneitä ongelmia.

Tutkimuksessa esitetään luokittelu ohjelmiston jatkokehitettävyysongelmille. Luokittelun antamaa tietoa syvennetään katsomalla tarkemmin koodikatselmuksissa löytyneitä jatkokehitettävyysongelmien tyyppejä ja niiden jakaumia. Lisäksi työssä tutkitaan ihmisten tekemiä jatkokehitettävyysarvioita tarkastellen: arvioiden yksimielisyyttä, ihmisten taustan vaikutusta tehtyihin arvioihin, merkittävimmiksi koettuja jatkokehitysongelmia sekä ihmisten arvioiden suhdetta lähdekoodimittareiden antamiin tuloksiin.

Työn tulokset osoittavat, että kevyen toiminnallisen testauksen jälkeen tehty koodikatselmointi löytää kolme jatkokehitettävyysongelmaa yhtä toiminnallista virhettä kohti. Aineistoa analysoidessa havaittiin uusi jatkokehitettävyysongelmaluokka jossa ohjelmistossa tehty ratkaisu täytyy ” ajatella uudelleen” ennemmin kuin ”organisoida uudelleen”. Vaikka tällaiset ongelmat luultavasti ovat tyypillisiä ohjelmistotuotannossa, emme ole tietoisia aiemmasta empiirisestä tutkimuksesta, joka olisi tunnistanut ja analysoinut niitä. Ihmisten arvioihin ohjelmiston jatkokehitettävyydestä vaikuttavat enemmän ihmisten asema organisaatiossa ja suhde arvioituun ohjelmakoodiin kuin ihmisten koulutus tai ohjelmistotuotannon työkokemus. Verrattaessa lähdekoodimittareita ja ihmisten arvioita havaittiin, että koodimittareiden avulla ei voida havaita kaikkia ihmisen tunnistamia jatkokehitettävyysongelmia.

Asiasanat ohjelmiston jatkokehitettävyysongelmat, refaktorointi, kontrolloitu koe, observointi, vastaajien välinen yhdenmielisyys, koodikatselmointi

| | | | |
|---|---|---|---|
| ISBN (painettu) | 978-951-22-9856-3 | ISSN (painettu) | 1795-2239 |
| ISBN (pdf) | 978-951-22-9857-0 | ISSN (pdf) | 1795-4584 |
| Kieli | Englanti | Sivumäärä | 66 s. + liitt. 80 s. |

Julkaisija Teknillinen korkeakoulu, Tietotekniikan laitos

Painetun väitöskirjan jakelu Teknillinen korkeakoulu, Tietotekniikan laitos

[✓ Luettavissa verkossa osoitteessa http://lib.tkk.fi/Diss/2009/isbn9789512298570/]

**ACKNOWLEDGEMENTS**

Completing this dissertation has been my main goal for the past five or six years, but my work and interest on the topic of software evolvability began even earlier. In the summer of 2001, I was engaged to serve the public at the Laboratory of Software Business and Engineering (SoberIT). My task was to maintain and further develop a measurement tool set that had been developed under the recently completed Lucos Tools research project. Certain parts of the tool set had high evolvability, but some did not.

First and foremost, I wish to express my gratitude towards this dissertation’s de-facto instructor and supervisor, Casper Lassenius, whose support, guidance, and wit has greatly helped this work.

During my dissertation work, I have participated in three SoberIT research projects: SEMS, SHAPE, and ESPA, which have been carried out by the Software Process Research Group (SPRG). Most of the work was done in the SHAPE project, some initial studies were made under the SEMS project, and the work was finalized in the ESPA project. I am very grateful to the “core” members of SHAPE research project team for their ideas, support and friendship during this undertaking: Juha Itkonen, Kristian Rautiainen, Jari Vanhanen, and Jarno Vähäniitty. Their individual contributions have been so numerous that I cannot possible list them all here. Additionally, I wish to thank all the individuals who have participated in the SPRG paper review meetings that have greatly improved this thesis.

I am also thankful to Reijo Sulonen, my official instructor and supervising professor, for his help during this work. I also wish to thank my pre-examination professors, Erik Arisholm and Tarja Systä, who helped to finalize this dissertation. I am grateful for the anonymous peer reviewers of the conferences and journals, whose comments have helped me to improve my research.

I would like to acknowledge the financial support of this project to the National Technology Agency of Finland (TEKES), the Department of Computer Science and Engineering, and the Graduate School on Software Systems and Engineering (SoSE).

Finally, I wish thank my father, Ilmari, and my mother, Sinikka, for their love and encouragement. I want to thank my two daughters, Irinja and Venja, who were born during the dissertation work, for their existence and for providing me with many moments of laughter and joy that have allowed me to take my mind off this work. Finally, special thanks to my spouse, Terhi, who has supported and loved me throughout this work.

## LIST OF PUBLICATIONS

This dissertation is based on the following publications, which are referred to in the text by roman numerals.

I. Mäntylä, M., Vanhanen, J, and Lassenius, C., "A Taxonomy and an Initial Empirical Study of Bad Smells in Code", in Proceedings of the *International Conference on Software Maintenance*, pp. 381-384, 2003, Amsterdam The Netherlands.

II. Mäntylä, M. V. and Lassenius, C., "Subjective Evaluation of Software Evolvability Using Code Smells: An Empirical Study", *Journal of Empirical Software Engineering*, vol. 11, no. 3, 2006, pp. 395-431.

III. Mäntylä, M. V., "An Experiment on Subjective Evolvability Evaluation of Object-Oriented Software: Explaining Factors and Interrater Agreement", in Proceedings of the 4th *International Symposium on Empirical Software Engineering*, 2005, Noosa, Australia, 10 pages in electronic proceedings

IV. Mäntylä, M. V. and Lassenius, C., "Drivers for Software Refactoring Decisions." in Proceedings of the 5th *International Symposium on Empirical Software Engineering*, pp. 297-306, 2006, Rio de Janeiro, Brazil.

V. Mäntylä, M. V. and Lassenius, C. "What Types of Defects Are Really Discovered in Code Reviews?", *IEEE Transactions on Software Engineering*, preprint 15 Aug 2008, <http://doi.ieeecomputersociety.org/10.1109/TSE.2008.71>.

The author of this dissertation is the primary author of all the included publications. He is responsible for all the research reported and for all the text in the publication excluding some editing of the text. The other authors have helped shape the argumentation of the publications and given instructions and improvement suggestion in the research described in the publications.

## TABLE OF CONTENTS

# 1. INTRODUCTION

## 1.1 Motivation and Background

Currently, software is found everywhere. Besides home computers, the Internet, telephone networks, and satellites, software is found in vehicles (cars, trains, airplanes, and ships), home electronics (phones, microwaves, dishwashers, cameras, and personal video recorders) factories (pulp mills, nuclear plants, and car factories), and buildings (elevators, escalators, access controls, and ventilation systems). Thus, modern society is heavily dependent on software and the organizations that make software.

*Software evolution* is the process of developing the initial version of software and the further development of that initial version to reflect the growing and changing needs of various stakeholders such as users, customers, company shareholders, programmers, and product managers. It has been long recognized that almost all large and successful software systems and products need continuous evolution. For example, in his 1975 book, Brooks [24] stated "the product over which one has labored so long appears to be obsolete upon (or before) completion. Already colleagues and competitors are in a hot pursuit of new and better ideas." and "As soon as one freezes a design, it becomes obsolete in terms of its concepts". Lehman, who studied system evolution at IBM in the 1970s, captured this idea in the first law of software evolution [74, 75], the law of continuing change, which states that a system needs to be continually adapted to reflect the changes in the real world.

Thus, for a system of products to be successful, continuous evolution is required. To make matters more complicated, the evolution of an individual software product or system can take several years. For example, this dissertation was created using Microsoft Word 2003 word processor, the eleventh commercial release from the evolution of the 25-year-old Microsoft Word product. Thus, evolution of a software system can last for decades and cost a lot of money. The *Seattle Times* [119] estimated[1] that the payroll costs alone of developing Microsoft's latest operating system, Vista, were $10 billion. The evolution of Windows Vista began in late the 1980s with the development of the Windows NT operating system. The total evolution cost of Windows Vista could be roughly $30 billion.

This study is about *software evolvability*, a quality attribute that reflects how easy software is to understand, modify, adapt, correct, and develop further (for further discussion on the term software evolvability, refer to section 1.2). In other words, software evolvability affects the costs of software evolution. For example, a one percent savings in the evolution of Windows Vista would result in the savings of $300 million. However, the economic impact of software evolvability is much larger than one percent.

Experiments by Bandi et al. [9] and Rombach [107] compared two functionally equal systems that had different levels of structural evolvability. Bandi et al. found that adding new functionality took 28% longer and fixing errors took 36% longer for the less evolvable system. Rombach's data indicated that requirements changes took 36% longer and error fixing took 28% longer in the less evolvable system. Studies utilizing industrial data do not have the luxury of comparing two functionally equal systems. Thus, they use regression models [10, 29, 77] to show that poor software structure is correlated with lower productivity and greater rework. Based on such models, Banker et al. [10] reported that software structure may account for up to 25% of the total costs occurring after the initial commercial release. It

[1] The *Seattle Times* had also asked for a figure of evolution costs from Microsoft's CEO Steve Palmer, but he could not provide such a number.

appears that differences in software structure evolvability can increase the cost of software evolution from 25% to 36%.

In addition to software structure, code layout and code documentation (naming and commenting) have also been shown to have an effect on program evolvability [53, 88, 99, 120]. Miara et al. [88] found that two and four space indentations were correlated with better program comprehension when compared with zero and six space indentations. Oman and Cook [99] discovered that an advanced form of code commenting and layout (called the book paradigm in the original work) is correlated with significantly higher comprehension test scores and slightly shorter test times when compared with traditional commenting and layout. The results of Tenny [120] indicated that code commenting has a larger effect on program comprehension than code structure. Evidence of the benefits of code element naming indicates that proper identifier name length is correlated with shorter debugging times [53]. Furthermore, studies of source code [93, 125] modifications have shown that code element renaming is frequently performed in practice. To summarize, the scientific empirical evidence seems to indicate that source code evolvability has a clear economic importance through its effect on feature development and error fixing efficiency.

Practitioners have also recognized that poor software evolvability has economic importance. For example, Microsoft's Office division has determined that 20% of the development effort should be budgeted to code modification [39]. An empirical study 11 years after the original study showed [73] that the actual time spent on making the code more maintainable was 15% from the development effort. Although this number is slightly smaller than what was stated earlier, it shows that the idea is implemented in practice. Extreme programming software development methodology [16] emphasizes short development cycles with frequent releases, close collaboration with customers, minimal up-front design, and willingness to adapt continuously to change. To be able to cope in such an environment, extreme programming emphasizes continuous refactoring as a method of assuring high evolvability. Industry experts [7, 49] advocating refactoring claim that high evolvability results in several benefits (Table 1). Finally, some industry experts view poorly evolvable code as technical debt that can slow development. Thus, debt should be promptly paid back to avoid high interest when working with poorly evolvable code [38]. Although the industrial sources mostly lack supporting data, it seems that industrial experts consider software evolvability important.

**Table 1. Why software evolvability is important - Motivation for refactoring** [7, 49]

| |
|---|
| • Improves software design [49] |
| • Makes programs easier to understand [7, 49] |
| • Helps locate bugs [7, 49] |
| • Increases software development speed [7, 49] |
| • Makes testing, auditing, and documenting easier [7] |
| • Reduces dependency on individuals [7] |
| • Increases job satisfaction [7] |
| • Extends system's lifetime [7] |
| • Preserves software asset value to organization [7] |

## 1.2 Terminology

In this work, the term *software evolvability* is a quality attribute that reflects how easy software is to understand, modify, adapt, correct, and develop further. Traditionally, this quality attribute is referred to as *software maintainability* that is defined by IEEE [64] as “the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or to adapt to a changed environment.” The word *maintainability* is derived from the verb *maintain*, which according to Merriam-Webster’s dictionary[2] is “To keep in an existing state (as of repair, efficiency, or validity); to preserve from failure or decline (maintain machinery).” Thus, the word maintainability refers to the ability to keep software in an existing state or to preserve it from decline. The problem with this definition is that software is not consumed or worn down by use. The term software maintenance offers a reasonably good fit with software project context where software is developed and, after the release, only bug fixes are made to the software. However, even in software project context, most software systems are subject to many changes after their initial deployment. Studies have also shown that 80% [103] of the maintenance phase changes are improvements to the existing system. The term software maintenance also offers a poor match with the development of software products such as Microsoft Word, or software services such as Google search engine. In software product and service development, the development is typically evolutionary and the current product is continuously improved to attract new customers and to keep existing customers satisfied. Therefore, based on reasons presented in this section, the term software evolvability was chosen over the traditional term software maintainability.

The relationships among the other relevant terms and their definitions are in Figure 1.

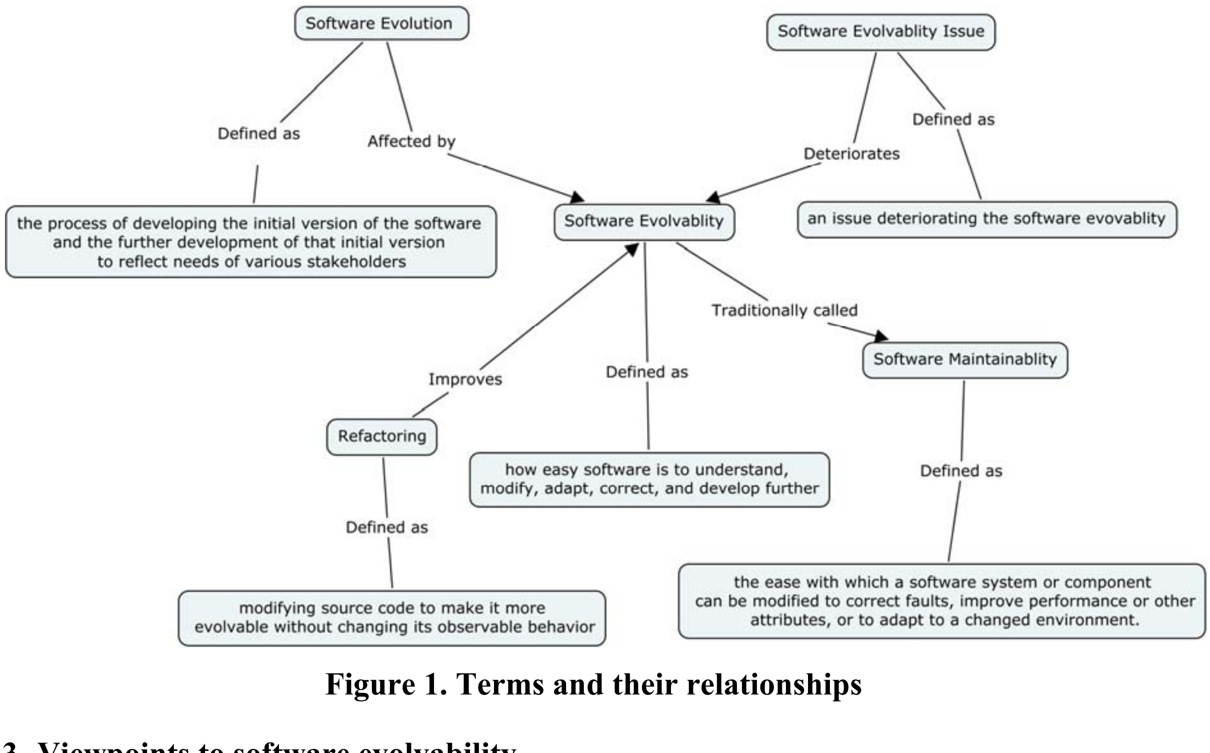
Software Evolution
Defined as
Affected by
the process of developing the initial version of the software and the further development of that initial version to reflect needs of various stakeholders
Software Evolvablity Issue
Deteriorates
Defined as
an issue deteriorating the software evovablity
Software Evolvablity
Improves
Defined as
Traditionally called
Software Maintainablity
Refactoring
how easy software is to understand, modify, adapt, correct, and develop further
Defined as
Defined as
modifying source code to make it more evolvable without changing its observable behavior
the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or to adapt to a changed environment.

**Figure 1. Terms and their relationships**

## 1.3 Viewpoints to software evolvability

Software evolvability is an internal software quality attribute and it cannot be directly experienced by the users of the systems. We have identified four viewpoints regarding

[2] http://www.m-w.com/

software evolvability (Table 2). First, we may study factors that affect software evolvability, e.g., why a piece of software has become poorly evolvable. The list of those factors is likely to be extensive, ranging from the programming language used and the motivation of the developers, to the business goals and organization of the developing company. Some work in this area has been done by Oman et al. [97], who listed several different factors affecting software evolvability. Lehman [75] has proposed laws that affect software evolution, some of which also affect software evolvability.

**Table 2. Viewpoints on software evolvability**

| |
|---|
| **Affecting factors**: Which factors can explain the current level of evolvability? |
| **Evaluation**: How can we evaluate software evolvability? |
| **Improvement**: How can we improve software evolvability? |
| **Effect**: What difference does evolvability make (e.g., in terms of development effort)? |

Second, we can look at how evolvable a piece of software currently is, and what features in the software contribute either positively or negatively to software evolvability. The evolvability evaluation can be studied at many abstraction levels, from requirements all the way up to the source code. Two main themes can be found in the evolvability evaluation: first, software engineering professionals have presented evolvability criteria, i.e., what they consider to be the characteristics of highly or poorly evolvable code and design [25, 33, 50, 78, 102, 115]. Second, several studies have proposed various code and design metrics that can be used to quantify evolvability criteria [19, 21, 30, 56, 58-60, 81, 85, 117]. The first theme can be thought of as human based evolvability evaluation criteria, while the second theme can be viewed as the automation of those criteria. Software evolvablity evaluation is further discussed, along with a definition of the scope of this work, in Section 1.4

Third, the improvement of software evolvability can be divided into at least three categories First, it is possible to affect the individuals that perform software development through training, personnel selection or other means. Second, it is possible to make improvements to the software development environment and organization – having a fully automated test harness with excellent test coverage for a large and complex system, for example, would improve evolvability because developers would have more courage to make modifications to the system and would spend less time testing their modifications. Finally, there is the technical improvement of software evolvability, which is often referred to by such terms as *restructuring*, *refactoring*, or *reengineering*. In some cases, even rewriting is used to improve evolvability. Based on the definitions of software restructuring [7] and refactoring [49], both terms essentially refer to a modification made to the internal software structure to make the software easier to understand and modify, without affecting the behavior of the external software. Reengineering [31] on the other hand, refers to the examination and alteration of software in order to reconstitute and implement it in a new form. Generally, reengineering is used to refer to major system alterations, whereas refactoring and restructuring mean small changes in the code. Often, improvement of software evolvability is not studied in isolation. Thus, in a reengineering context, improvement in the software evolvability is only one of several goals, and is a partial focus of this thesis.

Fourth, we can study the effects of the current state of software evolvability on external quality attributes such as development efficiency or the number of errors introduced by source code modification. This is perhaps the most widely studied viewpoint of software evolvability. Several well-constructed studies [9, 20, 77, 107] show that evolvability of current system predicts the future development effort. These studies are the motivators for this work

## 1.4 Scope

Referring to the previous section regarding the viewpoints of software evolvability, we will now examine at the scope of this work. Figure 2 shows the focus areas of this thesis. The main focus is human evaluation of source code evolvablity. This important topic has a limited amount of previous research.

Software evolvability evaluation is important because it assesses the need for evolvability improvement and shows what types of improvements should be made. However, it is evident that the evaluation is not superior to other viewpoints listed in Table 2 or showed in Figure 2. The evaluation can be performed at several levels. We can evaluate evolvability at the level of software architecture that consists of systems and sub-systems. We can study the evolvability at the sub-system level and look at the package and class design. We can delve still deeper into details and study the evolvability of individual classes and methods. The code level is important because it always exists and represents the actual software structure. With class design and architecture levels, the problem of the accuracy and correctness of class design and architecture descriptions is always present, and, in numerous real world situations, such descriptions have been completely omitted. However, we see that no evaluation level architecture, class design, or code is superior to others.

Under evaluation of software evolvability, we can identify two fundamentally different evaluation approaches: subjective evaluation that is performed by humans, and objective metric-based evaluation that is performed predominantly by program analysis tools. Human evaluation was chosen as the primary focus of this paper because ultimately, the developer decides whether software evolvability should be improved or not.

Finally, software evolvability evaluation results are affected by at least four factors that are depicted in Figure 2. In this thesis, the primary affecting factor being studied is software internals, i.e. software structure. Software internals should be the principal component that affects software evolvability, although other areas such as change scenario, personal background, and environmental support also affect evolvability.

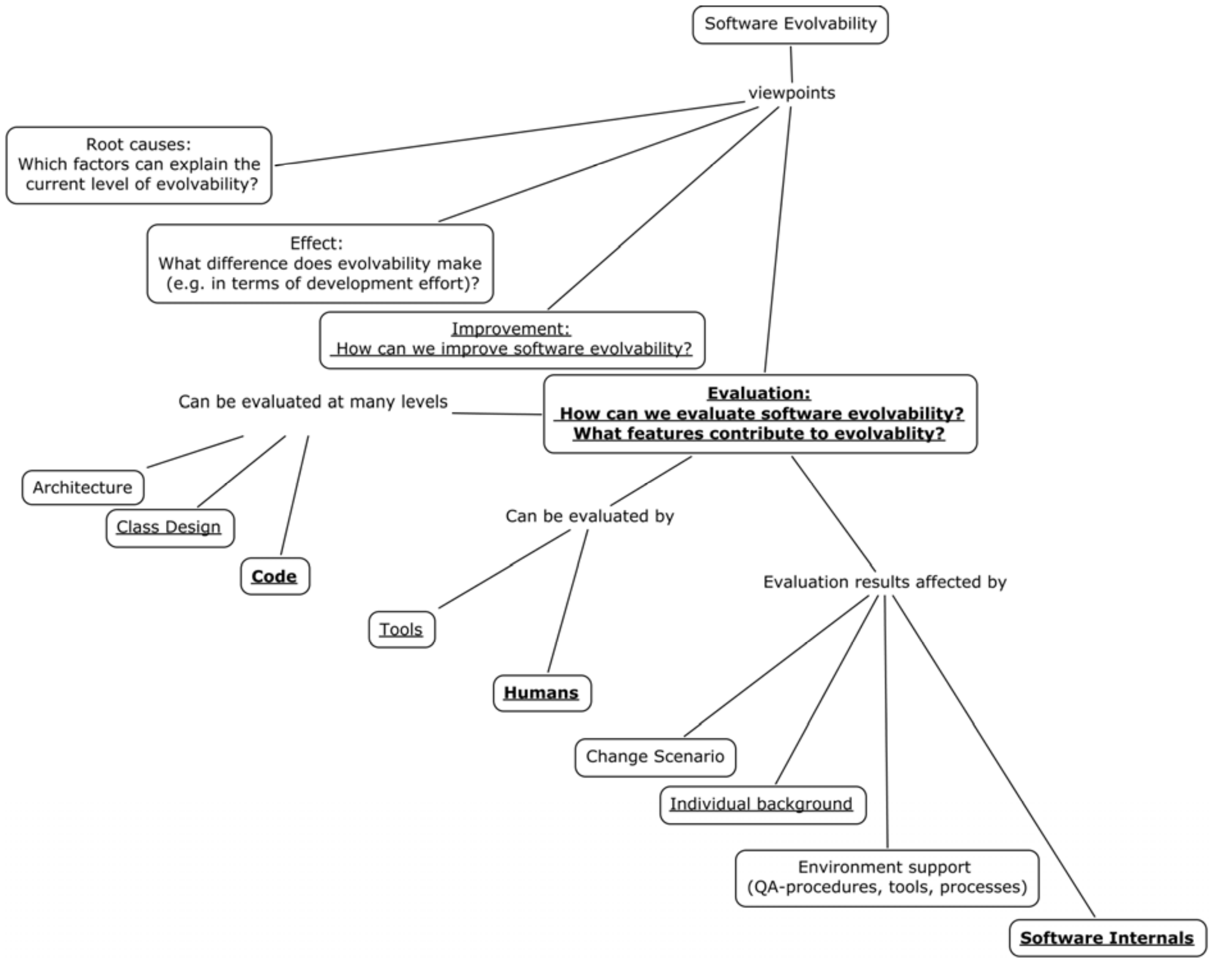
Software Evolvability
viewpoints
Root causes: Which factors can explain the current level of evolvability?
Effect: What difference does evolvability make (e.g. in terms of development effort)?
Improvement: How can we improve software evolvability?
Evaluation: How can we evaluate software evolvability? What features contribute to evolvablity?
Can be evaluated at many levels
Architecture
Class Design
Code
Can be evaluated by
Tools
Humans
Evaluation results affected by
Change Scenario
Individual background
Environment support (QA-procedures, tools, processes)
Software Internals

**Figure 2 Focus areas of the dissertation.
Primary focuses are written in bold and underlined. Partial focuses are underlined.
Other areas are out of scope.**

## 1.5 Research problem and questions

This dissertation consists of two research areas: software evolvability issues and human evaluations of software evolvability. Table 3 shows how the research areas and the research questions of this dissertation are linked to the publications of this dissertation.

**Table 3. Research areas and research questions**

| | **Research areas and research questions** | **Publication** | | | | |
|---|---|---|---|---|---|---|
| | | **I** | **II** | **III** | **IV** | **V** |
| | **Software evolvability issues** | | | | | |
| RQ 1.1 | How can the evolvability issues, either presented in the literature or identified by humans, be classified? | X | X | | X | X |
| RQ 1.2 | What types of evolvability issues are identified in the source code by humans and how are they distributed to different types? | | | | X | X |
| RQ 1.3 | What is the distribution of evolvability issues and functional defects found in code reviews | | | | | X |
| | **Human evaluations of software evolvability** | | | | | |
| RQ 2.1 | Do humans achieve interrater agreement when performing code evolvability evaluations? | | X | X | | |
| RQ 2.2 | Do the demographics of humans affect or explain the evolvability evaluations, and if so, how? | | X | X | | |
| RQ 2.3 | What is the relationship between evolvability evaluations and source code metrics, do the evaluations and metrics correlate or explain each other? | | X | X | X | |
| RQ 2.4 | What evolvability issues are seen as the most significant by human evaluators? | | | | X | |

### *1.5.1 Software evolvability issues*

Under software evolvability issues, three research questions were studied with qualitative research methods. First, we try to establish an understanding of software evolvability issues by classifying them. In many disciplines, but most notably in the natural sciences, classifications are created, maintained, and improved to increase scientific knowledge [96], e.g., Linnaeus's classification of the natural world [80] and the periodic table of chemical elements. Thus, the classification of evolvability issues has four purposes. First, we hope that classification will increase the body of knowledge in software engineering and increase understanding of the nature of software evolvability issues. Currently, there is a limited number of software evolvability issue classifications and even the existing classifications has limitations e.g. we see that the classification by Siy and Votta [113] is not descriptive enough. Second, the classification can be useful when creating company coding standards, code review checklists, and assigning roles to participant of code reviews. Third, the classification can be used as a basis for evolvability assessments (e.g., [1]). Fourth, the defect classification can provide input for creating automated defect detectors or developing new programming languages. For example, harmful code structures could be excluded from new programming languages (e.g., "goto" statements are no longer available in many languages).

The second research question under software evolvability issues considers the detailed evolvability issue types and their distributions. This research question deepens the understanding of software evolvability issues from a classification to a more detailed level. The distributions of the evolvability issues may reveal consistencies that hold over most software systems. However, the fluctuations can also be interesting since they may help explain the effect of the context to software evolvability issue types. The detailed evolvability issues types provide deeper understanding of the evolvability issues and may reveal new or ignored evolvability issue types. The first and second research questions complement each other in many ways. Thus, the motivations of the first research question mostly apply to the second research question.

It is important to study the detection mechanisms for evolvability issues. Therefore, the distribution between functional defects and evolvability issues detected in code reviews is considered in the third research question under software evolvability issues deals with. Prior work [113] has suggested that code reviews are a good method for detecting evolvability issues and it is vital to investigate this claim further. Making a distinction between functional defects and evolvability issues is important for three reasons. First, code reviews are often compared with various testing techniques that cannot detect evolvability issues, providing an incomplete picture of the benefits of code reviews. Second, code review literature has left the impression that code reviews are mainly about finding functional defects. Third, the practical implications of the distribution between functional defects and evolvability issues are significant. For example, one would expect software product companies to be interested in discovering evolvability issues, as their products may need to withstand heavy subsequent evolution, whereas software project companies would be less interested in evolvability issues.

### *1.5.2 Human evaluations of software evolvability*

The research area regarding human evaluations of software evolvability consists of four research questions that are studied using quantitative research methods First, we assess the *interrater agreement* of human evaluations of software evolvability. In practice, the assessment of evolvability issues is conducted by humans and can be seen as a sort of beauty contest of software, e.g., what seems like a nice and cleverly programmed functionality to one developer may seem like an ugly hack to another developer. Thus, the levels of the agreement of different types of evolvability issues are important to understand. This can help identify areas where humans achieve high agreement and areas where the reliability of human evaluations is debatable. This information can be used when deciding whether there is a need to double check evolvability issues found by individual developers.

Second, assuming that there are differences in the human evaluations, we wish to understand whether the demographics can explain the differences. For example, does experience or education make people more sensitive to evolvability problems? Such information can be used when placing evolvability evaluations in a context. For example, if we know that less educated developers are less likely to identify evolvability problems, there might be a risk of creating poorly evolvable software when using developers with limited educations.

Third, we wish to understand the relationship between evolvability evaluations and static code analysis with source code metrics. Source code metrics have been widely studied, but the link to human evaluations has only been partly explored. Understanding the link between source code metrics and human evaluations is important in determining whether code metrics can be useful in evolvability issue identification and refactoring decision mentoring. For example, if we know that people and static analysis tools have a high level of agreement on certain evolvability issues then it makes sense to use or develop tools for detecting such issues and use human effort for more creative and demanding tasks.

Fourth, using quantitative methods we study which evolvability issues are seen as the most significant by human evaluators. This can be seen as a partial return to the software evolvability issues research area. We try to quantify the importance of the evolvability issues by linking evolvability issues to the refactoring decisions. In other words, we wish to find out which evolvability issues are seen as the most severe. One should note that this severity is based on human opinions and it does not mean that such evolvability issues would have the highest economical impact. Thus, it only acts as a surrogate for better measures that were not available during the study.

## 1.6 The structure of the dissertation

This dissertation consists of six chapters. First, the introduction presents the background and relevant terms. Additionally, the research questions are briefly presented and the scope of the dissertation is outlined. Next, the literature review chapter presents the relevant literature that falls inside the scope presented in Section 1.4. Third, the methodology chapter presents the data collection and analysis methods of this study. Additionally, it further discusses and motivates the research questions. Fourth, the results chapter summarizes the results of this dissertation. Fifth, the discussion chapter discusses the results, answers the research questions, and highlights the limitations. Finally, the conclusion chapter presents the conclusions of this work and highlights directions for future research.

# 2. REVIEW OF THE LITERATURE

This chapter reviews the relevant literature for this dissertation. We cover the literature from the two research areas. First, we look at the literature of software evolvability issues. Second, we review the literature on human evaluations of software evolvability. To conclude the chapter, we highlight the gaps in existing studies.

## 2.1 Software evolvability issues

In the literature, criteria of software evolvability are presented from two viewpoints. Some authors have listed desirable features that increase software evolvability while others have listed undesirable features that decrease software evolvability. Because the undesirable and the desirable features are opposite ends of the software evolvability quality attribute, they are both discussed in this section.

It must be noted that in many publications, the authors do not state that a particular issue would affect software evolvability. Rather, they state whether such a feature is or is not desirable in the software. In this work, we have made the interpretation that if the issue mentioned by the author is not likely to directly cause external failure or other externally visible quality deviation, and is more likely to have an effect on software evolvability, then the issue is one of evolvability.

Next, we present prior works of the evolvability criteria at the class and code levels. It is difficult to make a clear distinction between class and code levels, and therefore we do not try to make a distinction between them. First, we look at the proposed criteria of software evolvability. Second, we see how people have tried to quantify the occasionally vague criteria. Third, the section presents the studies that have empirical investigated the benefits of the evolvability criteria. Finally, we look at empirical studies that have investigated the evolvability issues actually existing in software.

### *2.1.1 Software evolvability criteria*

This section reviews the literature of software evolvability criteria and summarizes the sources based on their origins.

Perhaps some of the oldest principles contributing to software evolvability are the ideas of *high cohesion* and *low coupling* presented in the 1970s [115]. High cohesion means that software modules should be composed of elements that are heavily related to each other. Low coupling means that relations between software modules should be minimal and simple.

Parnas [102] suggested the idea of *information hiding*. This means that every module should capture and hide an important design decision from other modules, i.e., decisions that are difficult or ones that are likely to change. Furthermore, Parnas stressed that simple interfaces revealing as little design information as possible are used to protect the hidden information. Parnas suggested that using the information hiding principle increases software changeability and comprehensibility, and aid independent module development. Decomposition should not be based on other factors such as performing the same type of functionality or being executed at the same stage in the program. We can see that the idea of information hiding has many similarities with low coupling.

The old design principles also exist in works that are more recent. Information hiding is currently supported in many programming languages through encapsulation mechanisms that can be used to enforce the information hiding of certain software elements such as data structures or routines. Class responsibility cards (CRCs) [14], a method for object-oriented class design, require that every class's responsibilities be identified. According to the authors, *responsibilities identify problems to be solved.* This is very similar to the idea of information hiding in which every module should hide and solve an important design decision. The law of Demeter by Lieberherr and Holland [78] states that any method *M* of an object *O* may only call methods that belong to *O* itself, to *M*'s parameters, to the objects *M* has created, and to *O*'s member objects. Thus, this is a rule that helps to minimize coupling between objects. Coad and Yourdon [33, 34] presented design principles such as: low coupling, high cohesion, clarity of design (i.e., consistent class naming and clearly defined responsibilities), and keeping classes simple. Bass, Clements, and Kazman [13] presented tactics to achieve high modifiability, and two of their three tactics are actually improved and elaborated versions of high cohesion and low coupling. Currently, the old design principles can be applied to lower level design as the abstractions of programming language has risen significantly since the 1970s.

Fowler and Beck [50] have come up with a term called *code smell* to help software developers recognize problematic code. These code smells are general descriptions of bad code, e.g., a long method or a large class, that are supposed to help software developers decide when code needs refactoring. The goal of code refactoring is to make the software easier to understand and/or extend. Thus, code smells can be easily seen as code level evolvability criteria.

Structures similar to code smells are described by Brown et al. [25], who discussed software anti-patterns. These anti-patterns describe problems from class to architectural levels. Some of them are similar to code smells, e.g., God class is equal to a large class smell. However, the scope of their work is wider than software evolvability since they also discuss problems in software processes, badly behaving developers, and many other areas.

Coding conventions [118] or software developer guides [62, 72, 86, 116] can also be seen as evolvability criteria as many instructions in such guides have the most significant effect on software evolvability. For example, McConnell [86] discussed several characteristics of high-quality software construction which consists of the principles of class and routine design; the properties of high-quality routines; reasons for creating a class; classes to avoid; and instructions on the design and implementation of class interfaces, encapsulation, inheritance, constructors, member functions, and data. Whereas Fowler and Beck, and Brown et al. focused mostly on code structure, coding conventions, and developer guides often have much wider focus. They may additionally give instructions on code layout, code file organization, code element naming, code commenting, and other issues not related to software evolvability directly, such as defensive programming, the usage of suitable tools, and various collaboration methods.

In Table 4, the evolvability criteria based on the type of source it originates from are grouped. We have three somewhat distinct sources. First, positive criteria, i.e., works presenting desirable features in code or design, are called *design principles*. Second, negative criteria, i.e., works presenting undesirable features, are called *evolvability issues*. Third, we have *practitioners' guides*, i.e., coding conventions or software developers' guides that try to summarize the first two sources into practical instruction on programming. Although, practitioners' guides have more items in their evolvability criteria, their ideas are less novel and there is less discussion of the evolvability criteria. Practitioners' guides also have a much larger focus than works presenting positive or negative criteria.

**Table 4. Source of evolvability criteria**

| | **Design principles** | **Evolvability issues** | **Practitioners' guides** |
|---|---|---|---|
| Examples | Low coupling, information hiding | Large class, duplicate code | No code file should exceed 2000 lines, method names should be verbs, create classes to model real world objects. |
| Sources | [13, 14, 33, 34, 78, 102] | [25, 50, 122] | [62, 72, 86, 116, 118] |

It must be noted that the evolvability criteria presented in this section is mostly based on personal opinions and experiences rather than research. This was true at least when the ideas were first introduced. Sections 2.1.3 and 2.1.4 discuss the empirical evidence of the evolvability criteria.

### *2.1.2 Quantifying the criteria*

This section reviews the literature that has focused on quantifying the evolvability criteria through code metrics.

Code and design metrics have been widely studied [19, 21, 30, 56, 58-60, 81, 85, 117]. One of the reasons for the high interest in code and design metrics has been the idea that they can be used to quantify software quality attributes such as evolvability, understandability, and complexity. Furthermore, there are specific metrics that help to follow the design principles of high cohesion, low coupling, and information hiding. Thus, code metrics studies can be seen as an attempt to quantify the evolvability criteria. According to Fenton and Pfleeger [47], internal product attributes can be measured in two dimensions: size and structure.

This paragraph introduces some size measures and discusses their relation to software evolvability. Line of code (LOC) is the most commonly used software size metric. Fenton and Pfleeger [48] refer to the work by Grady and Caswell [55] for the most widely accepted definition of a line of code. According to this definition, a line of code is any statement in a program except comments and blank lines (often abbreviated as NLOC, non-commented lines of code). This measure can be used as a quantification of whether a method or a class is considered too large. Halstead metrics [56] were one of the first metrics for trying to capture software size by other means rather than just counting lines of code [48]. The building blocks of the Halstead metrics are the number of unique operators and operands, and the total occurrences of operators and operands. From those building blocks, Halstead derived a wide range of different metrics that are currently seen by many authors as confusing and lacking theoretical or empirical basis [26, 48, 57, 124]. Halstead metrics building blocks are, in fact, a measure of program size and/or complexity. Despite its criticisms, empirical studies by Oman et al. [36, 37, 100, 123] found it a satisfactory predictor of software evolvability. Basic building blocks of Halstead metrics can be used like lines of code measure when evaluating software evolvability. We can easily present other size metrics that can be linked to software evolvability such as number of methods in a class and number of classes in a package.

This paragraph presents some structure metrics and focuses on metrics that are claimed to quantify some evolvability criteria presented in the previous section. McCabe's cyclomatic complexity measures the number of independent execution paths in a computer program [85]. McCabe originally suggested that a small cyclomatic number per program module increases the testability and understandability of a module. McConnell [86] listed complexity reduction as one of the reasons to create a method or class. Many authors have studied coupling metrics and they link directly to low coupling evolvability criteria. In the object-oriented metrics suite by Chidamber and Kemerer [30], coupling is defined as occurring between classes when methods of one class use the methods or instance variables of another class. To get a good overview of coupling metrics, see an article by Briand et al. [21]. Cohesion has also been widely studied and it directly links to high cohesion evolvability criteria. Again, a good survey of cohesion is measurement is presented by Briand et al. [19]. The information hiding design principle has also been studied in the measurement context [106]. However, likely due to the vague and subjective description of the principle, the breadth and the quality of the work does not come close to the work performed with coupling and cohesion measurements.

Code metrics, among other things, offer a way to quantify the evolvability criteria and enable automatic evolvability evaluation. Code metrics can be roughly divided into size metrics and structure metrics. However, as the code metrics are not the primary focus of this study, we have only presented the most relevant literature of this widely researched topic. For more information on this topic, we refer to cited articles and textbooks by Henderson-Sellers, Lorenz and Kidd, and Fenton and Pfleeger [48, 59, 81].

### *2.1.3 Empirical studies of the benefits of the evolvability criteria*

Most of the evolvability criteria presented in Section 2.1.1 is not based on empirical work, but rather on the opinions of the authors or in some cases larger consensus among a group of experts. Most of the individuals presenting the criteria are highly experienced and respected software engineers. Thus, attempting to validate their claims has offered researchers a starting point for empirical studies of software evolvability. This section reviews the empirical studies of software evolvability criteria. We look at the evolvability criteria for software structure, but also for other elements such as code commenting and layout. Furthermore, we look at empirical studies linking code metrics and software evolvability since code metrics can be seen as a quantification of the evolvability criteria.

Briand et al. [20] studied object-oriented design guidelines by Coad and Yourdon [33, 34] from the perspective of maintainability. Briand et al. used student subjects tasked with studying the design documents of two systems with different functionality and design. In the task, the students would mark the spots where changes would be needed in order to respond to requirement changes. They concluded that the system designed using Coad and Yourdon’s principles was easier to maintain than the other system.

Arisholm et al. [4] studied the changeability of two object-oriented systems with the same functionality but different design. They used student subjects whose task was to perform four programming tasks to an existing code. Surprisingly, it was found that the system with “bad” design was easier to understand and change. However, in a follow-up study made with industrial software consultants, it was found that the “bad” design was more maintainable to the junior developers and that the “good” design was more maintainable for the senior developers[3]. The implication is that evolvability is not only an attribute of the software but it is affected by the developer’s demographics as well.

Deligiannis et al. [42] studied the maintainability of two object-oriented designs where one design was suffering from a *god class* problem and the other one was not. A god class problem refers to a case when an object-oriented class hierarchy contains a single class that is too big and has too much functionality. The researchers had 20 student subjects who completed a modification task and answered a survey questionnaire. Based on the data, it appeared that the system without a god class problem was slightly easier to understand and maintain.

Darcy et al. [40] studied the structural complexity of software through concepts of coupling and cohesion. The researchers used 17 software engineers who completed perfective maintenance tasks. The study shows that cohesion and coupling alone had no effect on software evolvability. However, coupling and cohesion had a significant interaction effect on software evolvability. This is depicted in Figure 3. The figure shows that the smallest effort was required for systems that had high cohesion and low coupling as one would expect. However, the highest effort was not required for a system that had low cohesion and high coupling, which is surprising. The study results indicate that jointly, the evolvability criteria high cohesion and low coupling have positive effects on software evolvability measured in work effort.

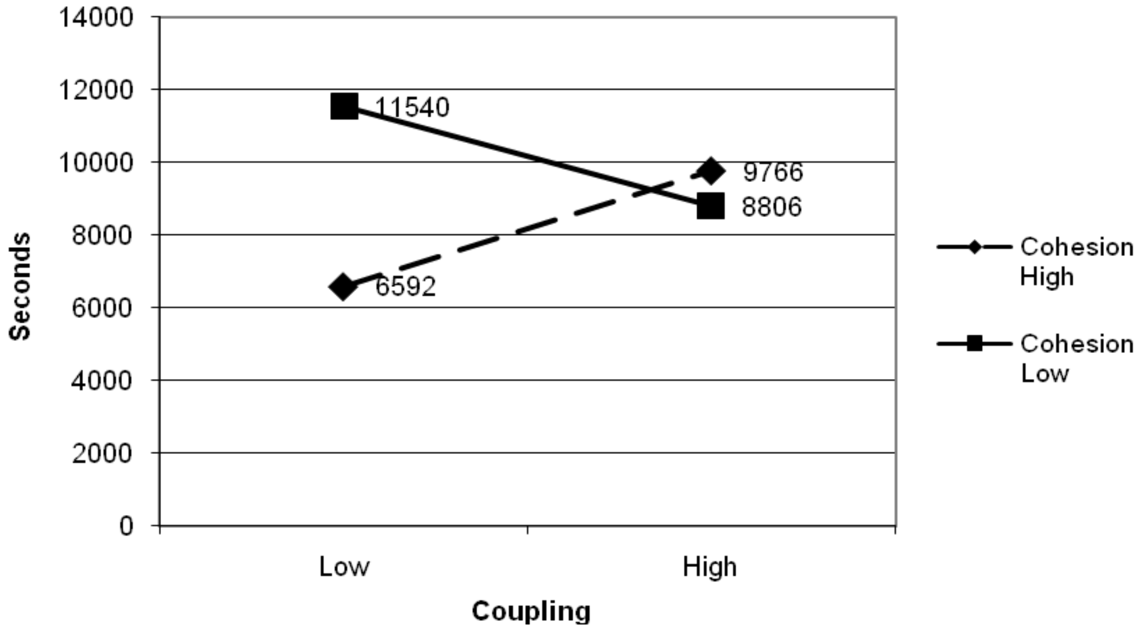

**Figure 3. Interaction effect of coupling and cohesion on effort** [40]

The works discussed thus far in this section studied design criteria with respect to software evolvability. In addition to pure design criteria studies, there are several works in which code metrics, which can be seen as surrogates for design criteria, have been used to predict the evolvability of software systems.

Rombach [107] studied source code metrics as maintenance effort predictors. The study used four systems, which varied from 1.5 to 15.2 KLOC in size, for which the researcher had planned identical maintenance tasks. Rombach's data indicates that requirements changes took 36% longer and fixing errors took 28% longer in the less evolvable system. The results propose with high significance that source code metrics can predict maintainability, comprehensibility, locality, and modifiability.

Li and Henry [76, 77] studied the correlation between maintenance effort and object-oriented metrics with two commercial systems. Maintenance effort was measured in a number of changed code lines per class. They concluded with high significance level that the chosen object-oriented metrics could be used as maintenance effort predictors. They also compared the object-oriented metrics and simple size metrics such as lines of code and number of methods. The comparison indicated that object-oriented metrics are better maintenance effort predictors than simple size metrics alone.

Chidamber et al. [29] studied the relation of object-oriented metrics and economic variables (productivity, rework, effort, and design effort) on three commercial software systems. The results show that high coupling and low cohesion are related to lower productivity, greater rework, and greater design effort after controlling the effects of individual developers and adjusting for size differences.

Bandi et al. [9] studied object-oriented code metrics and maintenance effort with university students. The study consisted of two different maintenance tasks that lasted between 90 and 120 minutes. The study results indicate that adding new functionality took 28% longer and fixing errors took 36% longer for a less evolvable system. The study showed with high significance that the used measures were useful maintenance effort predictors.

Arisholm [5] studied the impact of structural properties on the changeability of object-oriented software. The research found that none of the structural attribute measures was a significant predictor of change effort.

Additionally, there have been studies in which the structural properties of the software measured with code metrics, have been correlated with and used to predict defect counts [12, 18, 23, 46, 92, 127]. There have also been studies in which the structural properties of the software have been used to create effort prediction models [22, 95]. However, such studies are mostly out of the scope of this work as they focused on functional defects.

In addition to the structural properties of code, other factors affect software evolvability. Miara et al. [88] studied the affect of program indentation on program comprehensibility with experienced and novice subjects. The researchers found that the style of the indentation – blocked or nonblocked – made no difference. However, indentation level had a significant effect on program understanding. The highest test scores of program comprehension were received from programs with two-space indentation followed by four- and six-space indentation. The lowest scores were received from programs with no indentation at all.

Gorla et al. [53] studied debugging effort with a student experiment. They found that the lowest debugging times were associated with blank line share between 10-20%, and having 8-16 character long data item names.

Tenny [120] studied the use of comments and program readability with a student experiment. The research found that commented programs were easier to understand based on the number of right answers in a questionnaire that the students took during the experiment. Furthermore, the results of Tenny indicate that code commenting has a larger effect on program comprehension than code structure. Some practical implications of this were discovered by Lind and Vairavan [79] who found that program comment density is correlated with program development effort, indicating that developers used more comments for difficult code.

Oman and Cook [98, 99] presented a book paradigm for program documentation. The book paradigm combines a special program layout and program commenting in a format that is similar to traditional books. Their studies showed that the book paradigm aided program comprehension and reduced maintenance effort. Similarly, Arisholm et al. [6] found that code with UML-documents is more maintainable than code without UML-documents.

Table 5 summarizes the empirical research of the benefits of evolvability criteria. The table shows that there actually is empirical support for the proposed evolvability criteria discussed in Sections 2.1.1 and 2.1.2. However, whether the support is conclusive is debatable. For the first claim, there is a study indicating that evolvability criteria for software structure is not equal to all persons since developer experience affects the evolvability criteria. Additionally, for the first claim, there have been limited studies of bad code smell and anti-patterns. For the second claim, there appears to be conclusive evidence, although some studies do not support the claim. For the claims not focusing on software structure, there are limited studies supporting the ideas that proper code layout leads to higher evolvability.

**Table 5. Empirical evidence of evolvability criteria**

| Claim | Studies supporting | Studies not supporting |
|---|---|---|
| 1. Following evolvability criteria of **software structure** results in higher evolvability in terms of cost or effort | Briand et al. [20], Deligiannis et al [42], Darcy et al [40] | Arisholm et al. [4], Arisholm et al [3] |
| 2. Following evolvability criteria of **software structure quantified with code metrics** results in higher evolvability in terms of cost or effort | Rombach [107], Li & Henry [76, 77], Chidamber et al. [29], Bandi et al. [9] | Arisholm [5] |
| 3. Following evolvability criteria of **code layout** results in higher evolvability in terms of cost or effort | Miara et al. [88], Gorla et al. [53], Oman and Cook [98, 99] | |
| 4. Following evolvability criteria of **code commenting** results in higher evolvability in terms of cost or effort | Tenny [120], Oman and Cook [98, 99] | |
| 5. Following evolvability criteria of **code element naming** results in higher evolvability in terms of cost or effort | Gorla et al. [53] | |

### *2.1.4 Empirical studies of evolvability issues*

Only a limited number of studies exist that have presented empirical data of the actual reported evolvability problems. In other words, most of the evolvability problems, such as code smells by Fowler and Beck [50], or evolvability criteria, such as low coupling and high cohesion by Stevens et al [115], have not been empirically studied from the viewpoint of numbers and types of such evolvability problems. For example, we currently have little knowledge of the nature and the frequencies of the evolvability problems that are discovered in software.

Siy and Votta [113] studied the types of evolvability problems found in code reviews. In their study, only 18% of the findings identified were functional defects causing system failure, 22% were false positives, and 60% were evolvability issues. The research categorized 369 evolvability issues of 31 code inspection sessions and created four groups: *documentation* (share 47%), *style* (46%), *portability* (5%), and *safety* (2%). Furthermore, the documentation group had subgroups: *clarification* (need to add or improve comments), *correction* (comments were incorrect), and *documentation of future work* (refers to future development ideas). The style group had subgroups: *clean-up* (need for modifications to the code without changing the meaning of the program to prepare the code for subsequent evolution), *renaming* (renaming of code elements), *debugging* (need to add information on program execution to make debugging easier), and *cosmetic* (minor modifications, e.g., bracket conventions, indentations, blank lines). However, the study also had limitations. First, the study provided no detailed analysis of the defects, providing only high-level groups and their subgroups. Second, the classification is not descriptive enough. For example, the style group is described as issues related to an author’s personal programming style, which could be almost anything. In addition, the documentation group contains issues related to code commenting, but excludes code element naming; such issues are placed inside the style group. Code element naming and commenting should be in the same group, as they both communicate the intent of the code to the reader. Furthermore, having a safety group, meaning additional checks for scenarios that cannot possibly happen, for evolvability defects is unusual.

There have been studies focusing on the performed refactorings or problems detected in the source code. As refactorings are evolvability improvements, such studies are also empirical studies of evolvability issues in the source code. A Study of Eclipse IDE usage [93] found that renaming was used by all of the 41 developers participating in the study. It was followed by the *move method* and *extract method* refactorings that were both used by over half of the participants. Xing and Stroulia [125] studied source code changes in a software system. Their data shows that entity renaming was the most frequent refactoring followed by entity move and visibility change. Demeyer et al. [43] used code metrics to detect four type refactorings from three systems. The most frequent refactoring was split method followed by move method and split/merge subclass with roughly equal shares, and the most infrequent refactoring was spilt/merge super class. Additionally, there have been works studying duplication in a software system [8, 44, 121]. From Ducasse et al. [44], it was discovered that the amount of duplication varied from 6% to 25% in the different systems analyzed. The researchers note that in their study, systems with the highest duplication were purposefully chosen for the analysis as it was anticipated that they would contain high amounts of duplication. Thus, having one-quarter of duplicate code should be considered as an extreme value. The studies in this paragraph contain good quantitative data of the evolvability issues. Unfortunately, tools cannot detect all the issues a human would notice. Therefore, these studies offer a limited view of the evolvability issues that is restricted to the type of issues the analysis program is able to detect. Therefore, tool based analysis alone cannot be seen as a sufficient method for studying the types and distributions of evolvability issues.

To summarize the empirical studies of software evolvability criteria, we state the following. First, there has only been a single study, by Siy and Votta [113], that has holistically studied types and distributions of software evolvability issues. Second, there have been studies, which reveal frequencies of certain source code modifications, such as renaming, that are claimed to have positive effects on software evolvability. Third, there have been studies that measure the amount of some feature that make source code less evolvable, such as code duplication. We think that there should be more holistic studies looking at the software evolvability issues that are identified by humans as focusing only on a certain evolvability issues offers only a partial view.

## 2.2 Human evaluations of software evolvability

This section addresses the second research area: human evaluations of software evolvability. Here, we look at the prior empirical studies of human evolvability evaluations and their possible comparison to source code metrics or other automated analysis.

Shneiderman et al. (pp. 134-138 [112]) reported results from using peer reviews in software code quality evaluation. They conducted three peer review sessions that each had five professional programmers who had similar backgrounds and experiences as the participants. Each programmer provided one of his or her best programs that were then evaluated by the four other participants. The review was performed by answering 13 questions on a seven-point Likert scale. The questions varied from blank line usage and the chosen algorithm, to the ease of further development of the program. The results showed that in half of the evaluations, three out of four programmers agreed on the subjective evaluations (answers differed by one at the most). Still, in only 43.1% of the evaluations, the difference between all four evaluators was two or less. Thus, in over half of the evaluations, the difference between the minimum and maximum was three or more, indicating disagreement between the evaluators. The researchers tried to explain this by speculating that the subjects misunderstood the questions or the scale. However, the research does not account for factors such as differences in the developers' opinions about the program design, structure, and style that might explain the results.

Kafura and Reddy [67] studied the relationship between software complexity metrics and software maintainability. Maintainability was measured using system expert evaluations. Unfortunately, no details are given on how these evaluations were collected by the individuals and no data is provided of the evaluations. Nevertheless, the researchers conclude that the expert evaluations on maintainability were in conformity with the source code metrics.

Shepperd [111] validated the usefulness of information flow metrics on software maintainability by collecting the opinions of the maintainers for 89 modules of airspace software that totaled around 30,000 lines of code. Each maintainer was asked individually to classify each module from one to four on the perceived difficulty of some hypothetical maintenance task. In 73% of the individual classifications, the differences per module were one or less, and thus the researchers concluded that there was a strong correspondence between the individual ratings. Shepperd also found high correlation (0.7) between the subjective maintainability evaluations and information flow metric.

The Air Force Operation Test and Evaluation Center (AFOTEC) pamphlet [1] offers perhaps one of the most complete guides to performing human-based software maintainability evaluations. Its questionnaire part is partly utilized in the studies by Oman et al. [36, 37, 100, 123] and Muthanna et al. [94]. According to the pamphlet, the evaluation is performed by five evaluators who should have no relationship to the software to ensure that they are unbiased. As it is seldom humanly possible to evaluate an entire software system, the evaluation is performed on selected source code samples that are representative of the system. The evaluation is performed by agreeing or disagreeing, using a six-point ordinal scale, with statements that cover different aspects of software maintainability based on the source code and available documentation. Before the actual evaluation, a calibration run is made to ensure that the evaluators have a "uniform interpretation of how each statement applies to the system." However, the pamphlet particularly stresses that the evaluators should never be forced to change a score they have given. Thus, the purpose is to achieve agreement through discussion on the interpretation of the statements, while the answers are still allowed to vary between evaluators. After the calibration, the team proceeds to the actual evaluation. The statements are grouped into four categories: software documentation, module source listing, computer software unit, and implementation. Some example statements include program initialization is adequately described, identifier names are descriptive of their use, and dataflow in this unit is logically organized.

Oman et al. [36, 37, 100, 123] reported on the construction of a maintainability index. In this work, the researchers used source code metrics to create polynomial regression models that measured software maintainability. They calibrated the maintainability models based on how well they correlated with the subjective evaluations of the software maintainers of eight industrial software systems ranging from 1,000 to 10,000 lines of code [100]. After calibration, they performed a validation study in which they again acquired opinions and the source code on six industrial systems ranging from 1,000 to 8,000 lines of code. In the validation study, they also saw discrepancies where one engineer was more lenient and another more critical towards the systems they were evaluating. Although the study [100] does not directly indicate this, it appears that there was only the opinion of a single individual per software system that was used in the creation and the validation of the metric, which makes it difficult to effectively study the differences in human maintainability evaluation. After performing tests on several industrial systems, the researchers concluded that the automatic assessment corresponds well to the subjective view of the experts [123].

Muthanna et al. [94] used a similar approach as Oman et al. They started with 18 metrics, which they first narrowed down to six because most of the metrics had very high correlation. From the remaining six metrics, they chose three, which were found as the best maintainability predictors based on the developers' subjective opinions. From the three best maintenance predictor metrics, they created a regression model. They also validated the model in a single industrial software system with the size of 92 KLOC. In this validation, they compared model maintainability prediction with developers' opinions. In most cases, the prediction model was in line with developers' opinions, but there were also cases where it was not.

Kataoka et al. [69] analyzed a software system with a tool capable of detecting four simple evolvability issues (e.g., remove parameter, useless return value). The original developer of the software system analyzed the findings of the tool. The developer agreed with one third of the findings, disagreed with one third, and was undecided with the final third.

Kataoka et al. [70] studied the usefulness of improving software quality with refactoring and reported on a comparison between human evaluation and software metrics. According to the researchers, the subjective evaluation of an expert on the effectiveness of refactorings correlated quite well with measured improvement in the coupling metrics. The drawback in the study is that it consisted of only five refactoring cases and only a single developer evaluated the effectiveness.

Genero et al. [52] studied the maintainability of UML-class diagrams. The researchers showed that subjective evaluation of understandability, analyzability, and modifiability of UML-diagrams correlated with various class level metrics. In a follow-up study [51], they found a correlation between subjective complexity evaluation and both the time required to understand the UML-diagram (0,242), and objective code metrics based diagram classification (0,539). However, these studies are made with UML-diagrams and they lack results on the interrater agreement of the evaluations.

In recent work, Anda [2] compared expert judgment and structural measures in assessing software maintainability. The research found that the expert opinions and software metrics mostly corresponded to each. However, the research points out that several issues of software maintainability were not captured by the source code metrics.

## 2.3 Summary and gaps of existing work

Figure 4 illustrates the topics covered in this literature review and shows how our research questions link to these topics. Software evolvability can be, in the context of software internals (see Section 1.4 for details of scope of this work), operationalized with software evolvability criteria (see Section 2.1.1 for details of the criteria and Section 2.1.2 for details of its quantification). These criteria have been mostly created with expert opinions rather than empirical research of software systems. Furthermore, software evolvability issues, which are a subset of software evolvability criteria, have been studied less than the design principles (see Table 4 for details) which are also a subset of software evolvability criteria. Thus, our study focuses on increasing understanding about the human identified evolvability issues through empirical studies. We believe that this work can lead to improved software evolvability criteria, which can then again increase the benefits of applying these criteria. For current research on empirical benefits of software evolvability criteria, see Section 2.1.3. The only study, that the author is aware of, where researchers have holistically focused on human evolvability issues was done by Siy and Votta [113] who studied the types of evolvability issues identified in code reviews. Even that study did not contain a very detailed analysis of the evolvability issues (see Section 2.1.4 for a detailed description of that study).

The other research area of this study, human evaluations of software evolvability, was chosen because human evaluation plays a key role in software evolvability improvement. For example, if an individual does not recognize or consider a certain issue to be a problem, then that individual is not likely to remove this issue from the software. Therefore, differences in human evaluations can lead to difference in evolvability. Furthermore, this area has not been properly investigated. For example, little knowledge was available for assessing the reliability of the human evaluations. In many studies, referred in Section 2.2, human evaluations are not the primary focus of the study, but they are mentioned as a side note. Most of the studies also lack proper statistical analysis. Furthermore, many of the studies have been conducted prior to the era of object-oriented languages, which currently dominate the field of software development. Additionally, some of the recently suggested evolvability issues, i.e., the code smells by Fowler and Beck, lack empirical studies.

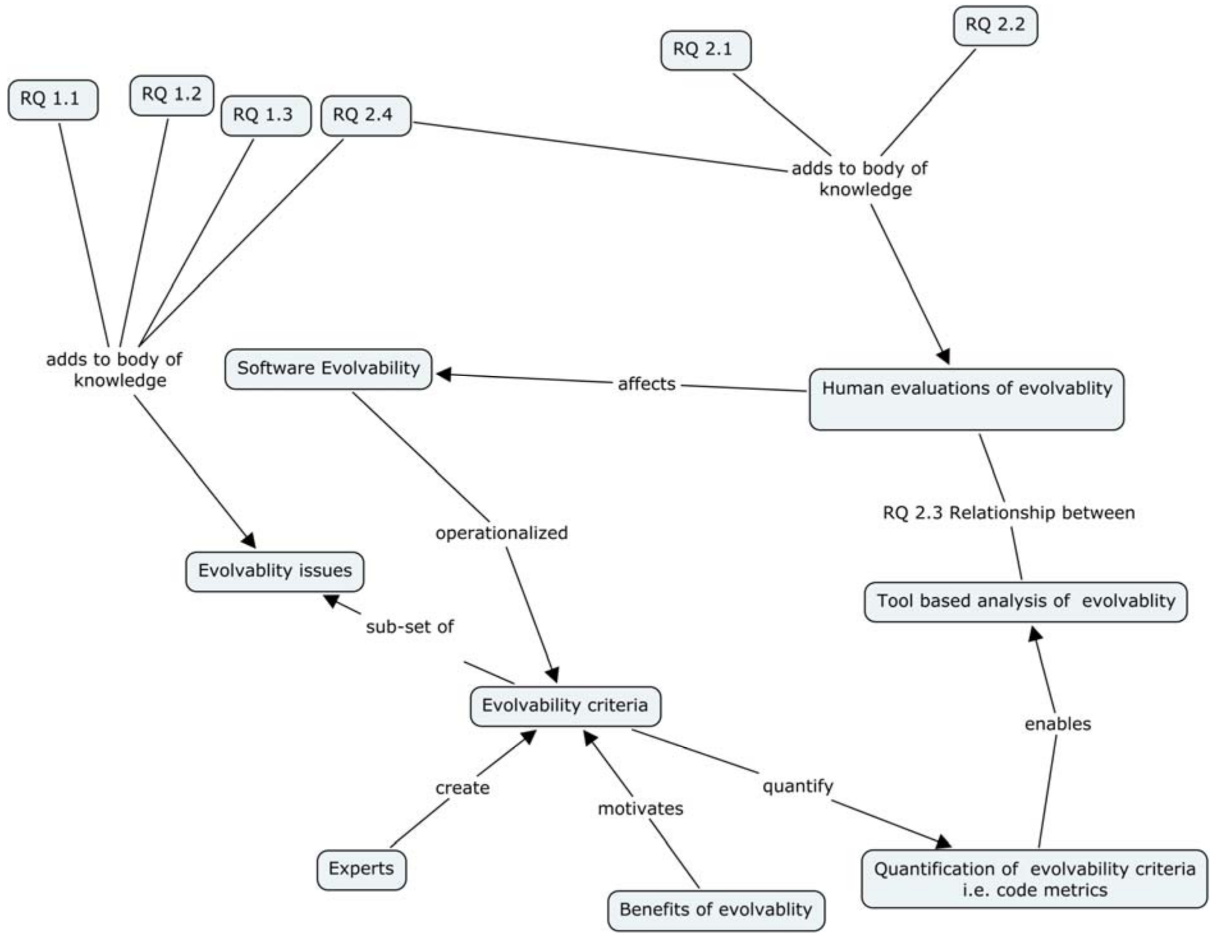
RQ 1.1
RQ 1.2
RQ 1.3
RQ 2.4
RQ 2.1
RQ 2.2
adds to body of knowledge
adds to body of knowledge
Software Evolvability
affects
Human evaluations of evolvablity
operationalized
RQ 2.3 Relationship between
Evolvablity issues
sub-set of
Tool based analysis of evolvablity
Evolvability criteria
enables
create
motivates
quantify
Experts
Benefits of evolvablity
Quantification of evolvability criteria i.e. code metrics

**Figure 4. Topics covered of existing literature and research questions.**

# 3. METHODOLOGY

This chapter presents the methodologies used in the studies. First, is an evaluation of the research approaches that were used in different parts of this dissertation. A brief presentation of the research questions and the methodology used follows.

## 3.1 Research approaches

This research consists of a series of studies that are reported in publication I-V. The publications are actually based on three studies that make up three separate data sources. The first study, which provides data for articles I and II, was performed in a software product company that had problems with evolvability. There, we studied the existence of code smells [50] through a survey of software modules that developers were most familiar with. After the survey, we gained a better understanding of the survey results through discussions with two lead developers. The first study had a number of limitations that were not possible to correct in a company setting, so we performed a second study. In that study, which provided data for articles III and IV, students evaluated the need for source code refactoring and the existence of certain code smells in a controlled experiment. The data collection for the second study was done twice, each with a different set of students. In both data collection rounds the student evaluated the refactoring needs of ten methods of a small software application. The biggest differences between the data collection rounds are as follows:

- The first round required evaluation of certain code smells for each method. In the second round we asked the rationales whether the code should be refactored or not. This was changed because we felt the rationales were more interesting than the evaluations of some predefined code smells.
- Data collection for the first round was done using survey forms in a controlled classroom setting where students had a pre-set amount of time to answer the questions concerning each method. In the second round, we used a web-based survey, and the students were free to use as much or as little time as they needed. However, in the second round we measured the time each student used to complete the survey. The time issue was changed between studies because the fixed time slot caused problems in the first round and we suspected that it might create even more problems in the second round when we asked for the rationales. We moved to a web-based survey because we felt that the extra control of a classroom setting was of little value compared to the extra effort it would require of the stakeholders in the experiment.
- In the first round, the students were given points if they simply took part in the experiment. In the second round, students were graded based on their rationales. We changed this because we wanted to encourage the students to provide their rationales for the refactoring decision.

The third study provided data for article V. In the second data collection round of the second study, we collected qualitative data of refactoring rationales that, for the most part, described evolvability issues. In the third study, we continued this qualitative study by observing and collecting data of code review defects. We observed industrial code review sessions and analyzed lists of defect data from students' code review sessions. We analyzed over 700 code review defects by their technical type and created a classification system.

Table 6 summarizes the research data of each article. In the table, the primary data source, data collection, and data analysis methods are presented. The data source is industry (I) or students (S); the data collection method is a survey (S), experiment (E), or observation (O); and the data analysis is either quantitative (N) or qualitative (L). Data collection in article IV was an uncontrolled experiment; the data was collected through a survey and is marked as S/E. Furthermore, N&L means that both qualitative and quantitative data analysis were major factors in the results.

One of the strengths of this dissertation is that the use of both student and industrial workers provides a nice variety of subjects and reduces the possibility of bias. The data collection methods could be improved. Specifically, there are weaknesses in the data collection methods of the first two articles. It appears that in those studies, it would have been more appropriate to use an interview as a data collection method or at least to interview the survey respondents in order to gain additional rationale for the answers. This would have provided a richer and more reliable data set for the first two articles. However, the data collection methods of the last three articles are suitable. The actual data analysis methods used were chosen based on the type of data that was collected, thus they have no major weaknesses.

**Table 6. Summary of research data**

| | **Article** | | | | |
|---|---|---|---|---|---|
| | **I** | **II** | **III** | **IV** | **V** |
| Data source | I | I | S | S | I |
| Data collection | S | S | E | S/E | O |
| Data analysis | N | N | N | N&L | L |

## 3.2 Software evolvability issues – classifications, types, and distributions

*RQ 1.1 How can evolvability issues, either presented in the literature or identified by humans, be classified?*

The philosophy of science divides classifications into logical and factual [96]. Factual classifications define classes based on some positive characteristics of the classified elements, e.g., the terms “male” and “female” create a factual classification whereas the terms “male” and “not-male” form a logical classification. Furthermore, it was noted by Niiniluoto [96] that factual classification are always more or less incomplete, but still improving and constructing such classifications is an important step in creating scientific knowledge. This terminology relies mostly on classifications done in real natural sciences, such as chemistry or biology (e.g., Linnaeus’s classification of natural world [80]). Qualitative research literature also recognizes the importance of classifications, although it is referred to with the term *clusters* by Miles and Huberman [89]. However, there are some differences, e.g., classification in natural sciences often do not allow overlapping classes, whereas qualitative studies may have overlapping clusters.

Classifications of evolvability issues are presented in articles I, II, IV, and V. All of the classifications are created based on similarities in the technical type of the studied evolvability issues. Thus, the classification is factual rather than logical. Furthermore, the classifications are closer to natural science classifications than qualitative science clusters as they describe real concrete constructs and they do not allow overlapping classes in principle.

The classifications were created based on analytical analysis of the evolvability issue types, i.e., the authors looked at the types of evolvability issues and created classifications based on similarities in the evolvability issues. In articles I and II, the classified evolvability issues were suggested by Fowler and Beck [50]. In articles IV and V, they were found by humans when studying the source code and then classified by the researchers. Furthermore, in article I, we performed a weak empirical verification for the validity of the classification based on the correlations between the issues.

*RQ 1.2 What types of evolvability issues are identified in the source code by humans and how are they distributed to different types?*

This research question is studied in articles IV and V. Qualitative data analysis methods were used to analyze the data [89, 91, 110], e.g., the coding process and the Atlas.ti program. Prior empirical work on the actual evolvablity issues found in the source code is limited, thus this research question was exploratory.

In the study described in article IV, the data came from an experiment in which 37 software engineering students evaluated methods in a Java program. The students were asked whether they thought the methods should be refactored or not, as well as to provide the rationale behind their decisions. The study analyzed the refactoring rationales to find what types of evolvability issues were identified. Additionally, the study performed statistical analysis using the SPSS program to link the identified refactoring drivers to the refactoring decisions, e.g., does a certain evolvability issue indicate higher likelihood in the refactoring decision. While writing the summary of this dissertation, the data of article IV was partly reanalyzed to make the results of the data analyzes easier to compare with the results of article V. The reanalyzed results can be seen in Section 4.2.

In the study described in article V, the data was gathered from two sources. Observations of industrial code review sessions provided the first data set. The second data set came from the defect logs of software engineering student code reviews that were performed as an exercise in one of our courses. Code reviews were studied because Siy and Votta [113] proposed that the majority of code review issues are actually evolvability issues. Thus, the study analyzed the types and ratios of issues that were found in the code reviews.

*RQ 1.3 What is the distribution of evolvability issues and functional defects found in code reviews?*

This research question is studied in article V. The data collection method for article V was described in the previous section. We wished to study the proposition by Siy and Votta [113] that most defects discovered in code reviews are evolvability issues rather than functional ones. This can help us understand the true benefits of code reviews and help to assess whether code reviews are a good tool evolvability evaluation and improvement.

### 3.3 Human evaluations of software evolvability

Research questions 2.1 through 2.4 involve human evaluations of software evolvability and are studied in articles II, III, and IV. In all articles, a questionnaire was used to collect evaluations of software evolvability. The study in article II was conducted in a software product company. The company developers evaluated the evolvability of the software modules with which they were most familiar. The evolvability evaluation was performed by evaluating the existence of the code smells by Fowler and Beck [50]. Article III is an experiment done with software engineering students. In the experiment, the students evaluated the evolvability of a small application that was developed by this author especially for the purposes of the experiment. The evaluation was performed at the method level; the students evaluated the existence of three code smells (long method, long parameter list, feature envy), and determined whether the method under evaluation should be refactored in order to make the software more evolvable. The study in article IV was also performed with software engineering students. The results of the experiment were also partially applied to article III. Details of article IV were already described in the previous section under RQ 1.2.

*RQ 2.1 – Do humans achieve interrater agreement when performing code evolvability evaluations?*

This research question is studied in articles II and III. When studying interrater agreement, we must first consider whether there should be interrater agreement between evaluators. When asking people for the best way to spend their holiday, we might not expect high interrater agreement, because people are likely to favor different things. However, we might expect high interrater agreement when we present the question to judges of ski-jumping contests. Thus, the purpose of interrater agreement analysis is to study the amount of agreement between the results of evolvability evaluations. In article II, this agreement analysis was studied by looking at the distribution and standard deviations of the evolvability evaluations. It is assumed that larger distribution and standard deviations indicate disagreement among the raters. Usually, distributions and standard deviations are not considered to be effective statistical methods when measuring agreement using the ordinal scale, but in article II, we had no choice because the number of participants did not allow better statistical methods to be used. In article III, more respondents provided evolvability evaluations; therefore, we used a robust statistical method, the Kendall coefficient of concordance noted as Kendall’s *W* [71], to study the interrater agreement. Kendall’s *W* gives a value that is between zero and one. A value of one indicates perfect agreement, while a value of zero indicates no agreement. Thus, the values of Kendall’s *W* resemble the values of standard correlation analysis such Pearson’s or Spearman’s correlation. To get reference points for values of Kendall’s *W*, we calculated its values using the first-round results of the ski-jumping World Cup competition that was held on December 29, 2004, in Oberstdorf, Germany. On that occasion, the five judges evaluating fifty jumps achieved Kendall’s *W* 0.888 and asymptotic significance p-value 0.000.

*RQ 2.2 – Do the demographics of humans affect or explain the evolvability evaluations and if so, how?*

This research question is studied in articles II, and III. In article II, we compared the differences between the evolvability evaluation in selected groups, e.g., developers and lead developers, more experienced and less experienced. The statistical analysis of the comparison is performed using the Mann-Whitney U test. This approach enables us to determine if there are differences in the evaluations among the selected groups. In article III, however, a different approach was adapted. In that study, we wanted to collect as much relevant demographic information from the respondents as possible in order to study which background factors, if any, have the strongest effect on the evolvability evaluations. This was studied using categorical regression that can be used for ordinal and nominal level data. Regression analysis can be seen as a more solid statistical approach than performing several Mann-Whitney tests that are likely to find at least some significant differences due to chance alone.

*RQ 2.3 – What is the relationship between evolvability evaluations and source code metrics; do the evaluations and metrics correlate or explain each other?*

This question is studied in articles II, III, and IV. In article II, this is studied using direct comparisons between the human evaluations and the suitable code metrics, e.g., human evaluation of the amount of duplicate code line versus the measured duplicated code lines. This approach is slightly problematic since human evaluations can be difficult to quantify in precise measures. In article III, categorical regression is used to predict the evolvability evaluations based on the selected code metrics. This approach allows us to find the metrics that are the best predictors for each type of evolvablity evaluation.  In article IV, we performed an analytical analysis to determine whether different evolvability issues that were found by humans could be identified with automatic static analysis tools or code metrics. This would determine if there are evolvability issues that cannot possibly be detected with either tools or code metrics.

*RQ 2.4 – What evolvability issues are seen as the most significant by human evaluators?*

This is studied in article IV. We created regression models that linked the identified evolvability issues (predictor variables) to the refactoring decision (predicted variable). This allowed us to see which issues that were identified in the source code were actually contributing towards the decision of whether or not the code needed to be modified. It is important to discern which evolvability issues are actually important and which ones are simply pointed out because people were asked to find evolvablity issues in the first place.

# 4. RESULTS

This section presents the results of this dissertation. The first section presents the classification of the software evolvability issues. The second section shows the detailed types and distributions of human identified software evolvability issues. The third section studies the amount of evolvability defects detected in code reviews in comparison to functional defects. The final section studies human evaluations of software evolvability.

## 4.1 Classification of software evolvability issues

This section presents our classifications of software evolvability issues. We start in the first subsection by presenting a classification based on the code smells by Fowler and Beck [50]. The original articles I and II describe this work in more detail. Then, at the second subsection, we look at classification created when analyzing human identified evolvability issues. This is covered in more detail in articles IV and V. Finally, in the last subsection, we combine the classifications. This combined classification is not present in the published articles. It can be seen as an extension to two prior classifications and an attempt to combine the two partly different classifications schemes.

### *4.1.1 Classification of code smells*

The first version of the code smell classification was presented in article I. In this section, we present the latter version of the classification that was presented in article II.

In the classification, there are five groups: *bloaters, object-orientation abusers, change preventers, dispensables,* and *couplers*. Additionally, two smells could not be placed inside any of the taxonomy's groups.

The bloater smells are *long method, large class, primitive obsession, long parameter list,* and *data clumps* (for description of code smells see Table 7 or [50]). Bloater smells represent something that has grown so large that it cannot be effectively handled. It seems likely that these smells grow a little bit at a time.

The object-orientation abusers are *switch statements, temporary field, refused bequest,* and *alternative classes with different interfaces*. The common denominator for the smells in the object-orientation abuser category is that they represent cases in which the solution does not fully exploit the possibilities of object-oriented design.

The change preventers are *divergent change, shotgun surgery,* and *parallel inheritance hierarchies*. Change preventers are smells that hinder changing or further developing the software. These smells violate the rule suggested by Fowler and Beck pp. 80 [50], which states that classes and possible changes should have a one-to-one relationship.

The dispensables are *lazy class, data class, duplicate code, dead code,* and *speculative generality*. The common thing for the dispensable smells is that they all represent something unnecessary that should be removed from the source code.

The couplers are *feature envy, inappropriate intimacy, message chains,* and *middle man*. This group has four coupling-related smells. One design principle that has been around for decades is low coupling [115]. This group has three smells that represent high coupling. The middle man smell on the other hand represents a problem that might be created when avoiding high coupling with constant delegation.

**Table 7. Description of bad code smells** [50]

| |
|---|
| **Long method** is a method that is too long, so it is difficult to understand, change, or extend. |
| **Large class** means that a class is trying to do too much and it has too many instance variables or methods. |
| **Primitive Obsession** smell represents a case where primitives are used instead of small classes. For example, to represent money, programmers use primitives rather than creating a separate class that could encapsulate the necessary functionality like currency conversion. |
| **Long parameter list** is a parameter list that is too long and thus difficult to understand |
| **Data clumps** smell means that software has data items that often appear together. Removing one of the group’s data items means that the those items that are left make no sense, e.g., integers specifying RGB colors. |
| **Switch statements** smell means a case where type codes or runtime class type detection are used instead of polymorphism. In addition, type codes passed on methods are an instance of this smell. |
| **Temporary field** smell means that class has a variable that is only used in some situations. |
| **Refused bequest** smell means that a child class does not fully support all the methods or data it inherits. |
| **Alternative classes with different interfaces** smell means a case where a class can operate with two alternative classes, but the interface to these alternative classes is different. For example, a class can operate with a ball or a rectangle class, and if it operates with the ball, it calls the method of the ball class playBall() and with the rectangle it calls playRectangle(). |
| **Parallel inheritance hierarchies** smell means a situation, where two parallel class hierarchies exist and both of these hierarchies must be extended. |
| **Lazy class** is a class that is not doing enough and should be removed. |
| **Data class** is a class that contains data, but hardly any logic for it. Classes should contain both data and logic. |
| **Duplicate code**. Duplicated code |
| **Speculative generality** Unnecessary code has been created in anticipating the future changes of the software. Predicting the future can be difficult and often this just adds unneeded complexity to the software. |
| **Message chains** smell means a case, where a class asks an object from another object, which then asks another and so on. The problem here is that the first class will be coupled to the whole class structure. To reduce this coupling, a middleman can be used. |
| **Middle man** smell means that a class is delegating most of its tasks to subsequent classes. Although this is a common pattern in programming, it can hinder the program if there is too much delegation. The problem here is that every time you need to create new methods or to modify the old ones, you also have to add or modify the delegating method. |
| **Feature envy** smell means that a method is more interested in another class or other classes than the one where it is currently located. This method is in the wrong place and should be moved. |
| **Inappropriate intimacy** means a smell where two classes are too tightly coupled with each other. |
| **Divergent change** smell means that one class needs to be continuously changed for different reasons, e.g., we have to modify the same class whenever we change a database, or add a new calculation formula. |
| **Shotgun surgery** smell is the opposite of the Divergent Change. It means that for every small change we must modify a bunch of classes, e.g., whenever we change a database we must change several classes. |

### *4.1.2 Empirical validation of the smell classification*

In order to validate the smell classification, we analyzed the correlations between smells in article I. However, because there have been changes in the smell classification since article I was published, we have re-drawn Figure 5, which shows the strongest ($r > 0.575$) and the most significant ($p < 0.01$) correlations between the smells.

It seems natural that the existence of some smells would correlate positively with some other smells while others would have a negative correlation. The taxonomy helps to capture strong correlations within groups. Figure 5 shows that 7 correlations are within the proposed groups but there are 98 correlations between the actual groups. Since we had 23 smells in our survey, this results in 253 correlations between all the smells. The total number of correlations within all groups is 36. The total number of between-group correlations is naturally 217 (253-36 = 217). This means that 19.4% (7 out of 36) of the total within-group correlations are strong, whereas only 4.15% (9 out of 217) of between-group correlations are among the strongest. These results seem to indicate that the taxonomy is also supported by the correlations between the code smells.

We have no further details or qualitative data regarding the nature of the correlations. Therefore, we can only speculate as to the nature of the correlations. For example, it is likely that large classes are built from long methods. Furthermore, it is quite possible that parallel inheritance hierarchies consist of lazy classes that are not doing much, since duplicated class hierarchy exists. Still, parallel inheritance hierarchies might lead to refused bequests where class is not supporting or using everything it has inherited. However, the nature of the correlation is only speculation and has not been empirically validated. Naturally, in such a large sample, some correlations will simply be due to chance.

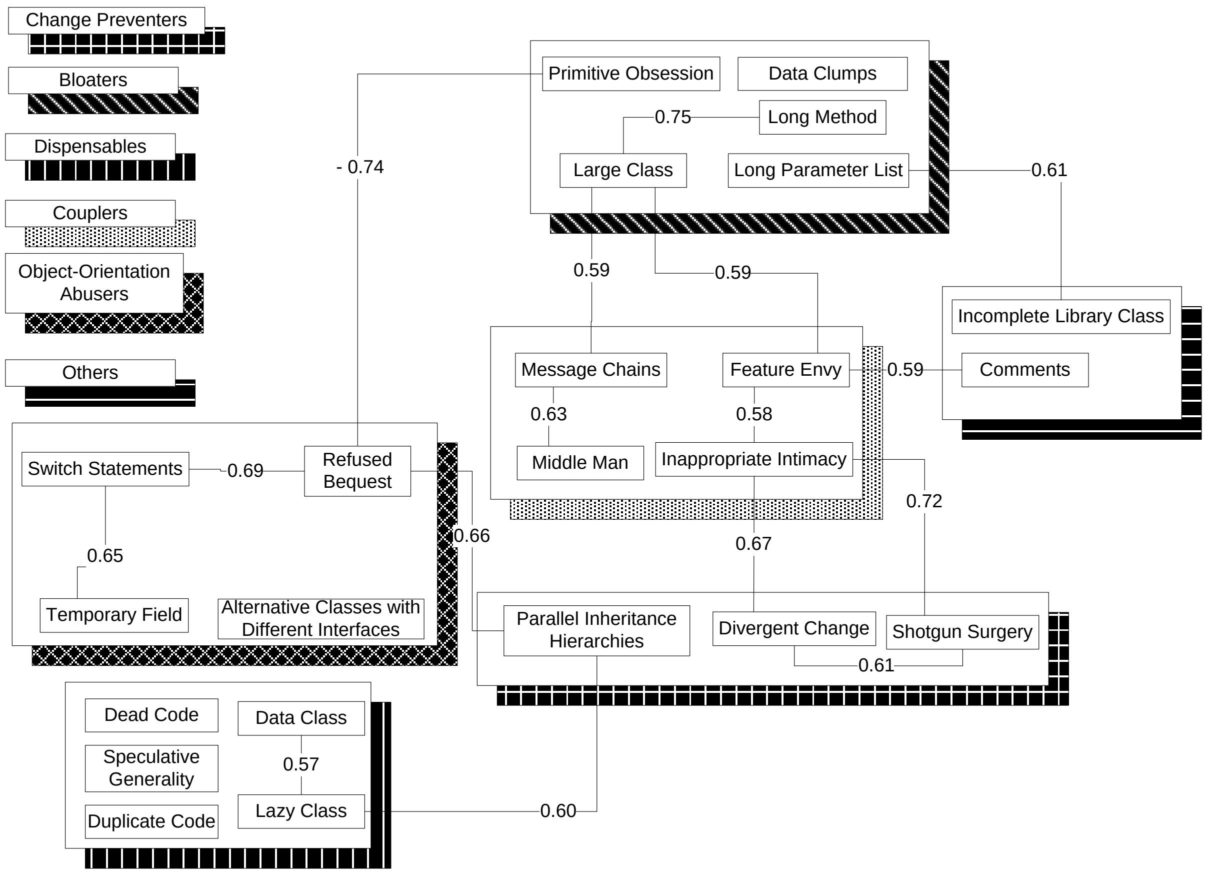

**Figure 5. Spearman correlations between smells (r > 0.58 and p < 0.01) and the smell taxonomy**

### *4.1.3 Classification human identified evolvability issues*

Article IV studied a human's refactoring drivers when evaluating ten Java methods. The drivers were separated into *negative* and *positive* driver groups. Negative drivers represent code problems leading to refactoring. They were further divided into the driver groups *documentation, visual representation, structure and general*. Naturally, one could also create subgroups of the positive aspects. However, in that study, only a limited number of positive comments could have been grouped. Most of the positive comments were often very short and very general indicating only that the method was acceptable and that there was no need for changes. Thus, it was not seen as beneficial to create subgroups of the positive comments.

Documentation means information in the source code that communicates the intent of the code to humans, e.g., commenting and naming of software elements like variables, functions, and classes. Visual representation means defects hindering program readability for a human eye. Structure stands for the source code composition that is eventually parsed by the compiler to a syntax tree. Structure is clearly distinguishable from documentation and visual representation, because the latter two have no impact on the program runtime operations or the syntax tree generated from the source code. Finally, the group general contained all negative comments that were too general to be placed in any of the other groups.

Article V studied code review findings of nine industrial and 23 student code reviews. That study extended the classification evolvability issues of Article IV. The documentation group was further divided in to the subgroups *textual* and *supported by language*. Supported by language issues are embedded in and enforced by the programming language, e.g., declaring an immutable variable or limiting the scope of a method. Textual documentation issues are those textually documented through naming and code commenting. Traditionally, documentation defects have only excluded the documentation defects that are supported by the language. Thus, one may question our decision. However, we think that if defects that are supported by the language would be moved under structure, the second most suitable group, this would make comparison between development languages even more difficult, since then such defects would be under documentation or structure, depending on the programming language.

In Article V, the structure group was divided into subgroups: *re-organize* and s*olution approach*. In article IV, the solution approach issues were present, but they were all placed inside a single driver called *Poor Algorithm.* The Reorganize subgroup consists of issues that can be fixed by applying structural modifications to the software. Moving a piece of functionality from module A to module B is a good example. Solution approach issues propose an alternative way of implementation that often requires only a limited amount of structural modification. For example, replacing a program’s array data structure with a vector, and knowing the existence of prebuilt functionality that can be used instead of using a self-programmed implementation, would be considered a solution approach issue. Therefore, solution approach recommendations are not about reorganizing existing code, but they refer to rethinking the current solution and implementing it in a different way. Thus, coming up with a solution approach is likely to require programming wit, out of the box thinking, and good knowledge of the development environment and the applied development practices. Reorganize issues, on the other hand, can often be found without deep knowledge of the development environment and practices, simply by assessing the code under review.

### *4.1.4 Combined classification*

Based on the classifications presented in articles I, II, IV, and V, this section presents a combined classification shown in Figure 6. From this final classification, we have removed the evolvability issue group General because it does not give any detailed information about the evolvability issues. Rather it expresses that the evolvability issue description is poorly written and cannot be effectively further categorized.

From the figure, we see that four out of five code smell groups belong under the reorganize group because they focus on structural modifications. The only group under the solution approach is object-orientation abusers, which represent implementation solutions that are not preferable when using object-oriented programming. One could also argue that change preventers could be placed under alternative approach. However, fixing such problems, moving functionality so that code affecting similar issues is inside the same class, is closer to reorganizing code than having a different solution approach to it.

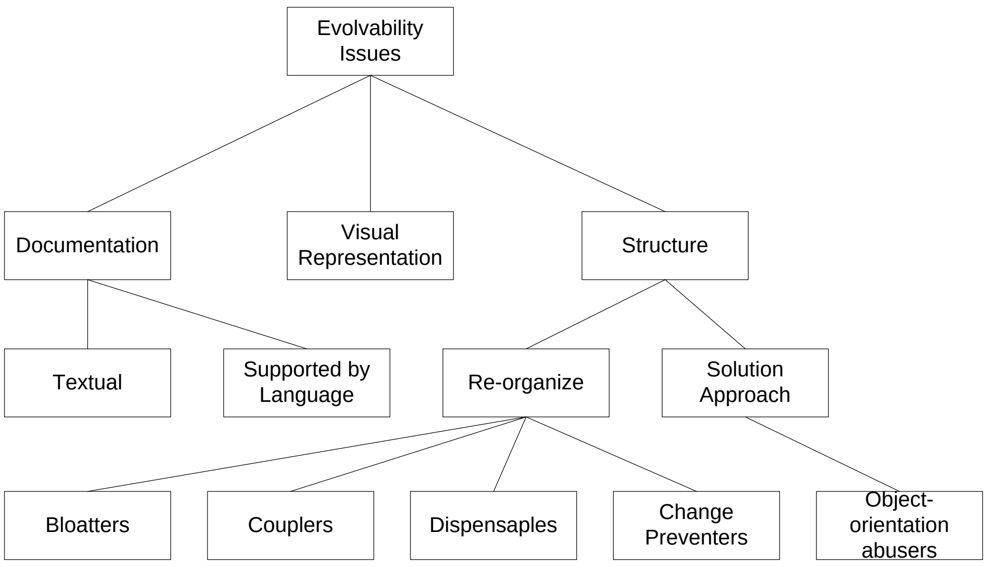

**Figure 6. Combined classification of the evolvability issues**

## 4.2 Types and distributions of software evolvability issues

In this section, we have combined the evolvability issues reported in articles IV and V. The numbers and distributions of the evolvability issues are not completely comparable between the studies. However, since both studies focus on evolvability issues identified in the source code, it makes sense to present them together. In Table 8 through Table 14, industrial reviews and student reviews refer to the evolvability issue counts in article V and refactoring experiment refers to the number of student answers identifying refactoring drivers in paper IV. The refactoring experiment numbers are not unique evolvability issue counts because several students might have identified the same issue. Furthermore, if a student identified, for example, three documentation issues from a particular method, this increased the count of documentation issues by only one because study IV has counted the number of students identifying particular type of issues. Therefore, the total numbers in the refactoring experiment column do not add up.

In Table 8, the shares of evolvability issues in different contexts are presented. In the reviews, there were 276 and 287 evolvability issues, and in the refactoring experiment, 245 out of the 360 student answers identified the need for evolvability improvements. In both reviews, approximately 10% of the evolvability issues were visual representation issues. The share of visual representation issues was only slightly higher in the refactoring experiment with about 15%. Roughly one third of the evolvability issues of industrial reviews were concerned with the documentation of the code. In the student reviews, almost half of the evolvability issues came from the documentation group, but in the refactoring experiment only a quarter of the answers identified a need for documentation improvement. In the industrial reviews, 55% of the evolvability issues belonged to the structure group, while in the student reviews, the percentage was only 43%. However, in the refactoring experiment, over 75% of the answers proposed structural modifications. In the refactoring experiment, there were 54 incidents where students gave negative comments from the method, but they were too general or too vague to be placed inside any of the groups. Such issues are seen as poor evolvability issue descriptions and they are not analyzed further in this study.

It is difficult to find any consistencies in the shares of the evolvability issues in Table 8, outside of the share of visual representation issues. It is possible that the high share of structural issues in the refactoring experiment was caused by the experiment design that particularly stressed the structural reasons for refactoring. Additionally, in the refactoring experiment, all the subjects analyzed the same code while in the reviews there were several source code documents. Thus, a possible bias of having additional structural issues in the refactoring experiment code may also have caused the radical differences in the shares of structural issues when compared to the code reviews.

**Table 8. Distribution of evolvability issues**

| Type | Industrial Reviews | | Student Reviews | | Refactoring experiment | |
|---|---|---|---|---|---|---|
| Documentation | 96 | 34.8% | 132 | 46.0% | 62 | 25.1% |
| Visual Representation | 27 | 9.8% | 31 | 10.8% | 35 | 14.7% |
| Structure | 153 | 55.4% | 124 | 43.2% | 190 | 75.7% |
| General | 0 | 0% | 0 | 0% | 54 | 25.9% |
| **Total** | 276 | 100.0% | 287 | 100.0% | 245 | 100% |

In Table 9, we can see that textual defects are distributed to two major types: naming and comments. *Naming* indicates poor names for software elements such as variables and routines, while *comments* means there is a problem with code comments, such as not having enough comments or having incorrect or unnecessary comments. The majority of the textual issues were related to code comments or code element naming with the latter being more frequent in the industrial reviews and the former being more frequent in the student reviews and the refactoring experiment. In discussions with the participants of the industrial reviews, it was clear that the company strongly believed in self-descriptive naming over code commenting, which explains the difference. *Debug info* contains information that is intended for a programmer who is debugging the software. Debug info is included in this group since it improves programs static and runtime documentation.

**Table 9. Documentation Textual issues**

| Type | Industrial Reviews | | Student Reviews | | Refactoring experiment | |
|---|---|---|---|---|---|---|
| Naming | 45 | 68.2% | 30 | 27.5% | 27 | 44.3% |
| Comments | 16 | 24.2% | 65 | 59.6% | 43 | 70.5% |
| Debug Info | 1 | 1.5% | 13 | 11.9% | 0 | 0% |
| Others | 4 | 6.1% | 1 | 0.9% | 0 | 0% |
| **Total** | 66 | 100% | 109 | 100% | 61 | 100% |

In Table 10, issues that were supported by language are listed. Code element type (e.g., int, boolean, string), code element immutability (const, final), and code element visibility were the three most recognized issues. Article IV does not report any such issues because few of them were present in that data set. It may be that such issues were seen as too trivial or minor to be considered as refactoring rationale so the students did not report them in the experiment. Another possible reason could be that there was a limited number of such issues present in the code that was used in the experiment. The higher number of the issues supported by language in the industrial reviews can be partly explained by the company's coding standard that required certain issues such as const usage (declaring a variable to be immutable) if the variable was not modified.

**Table 10. Documentation Supported by Language issues**

| Type | Industrial Reviews | | Student Reviews | | Refactoring experiment | |
|---|---|---|---|---|---|---|
| Element Type | 11 | 35.5% | 12 | 52.2% | 0 | 0% |
| Immutable | 13 | 41.9% | 2 | 8.7% | 0 | 0% |
| Visibility | 5 | 16.1% | 6 | 26.1% | 1 | 50% |
| Void Parameter | 2 | 6.5% | 0 | 0% | 0 | 0% |
| Element Reference | 0 | 0 | 3 | 13.0% | 1 | 50% |
| **Total** | 31 | 100% | 23 | 100% | 2 | 100% |

In Table 11, visual representation issues are presented. *Blank line usage* was the most frequent issue in the industrial reviews and in the refactoring experiment. In the student review, bracket usage was the most frequent issue. Distribution of issues was most even in the industrial reviews and had the most variation in the refactoring experiment. Furthermore, no space usage issues were identified in the student reviews or in the refactoring experiment, thus one may speculate that the students were less prudent than the industrial reviewers.

**Table 11. Visual Representation issues**

| Type | Industrial Reviews | | Student Reviews | | Refactoring experiment | |
|---|---|---|---|---|---|---|
| Bracket Usage | 5 | 18.5% | 11 | 35.5% | 3 | 8.57% |
| Indentation | 6 | 22.2% | 5 | 16.1% | 4 | 11.43% |
| Blank Line Usage | 8 | 29.6% | 7 | 22.6% | 24 | 68.57% |
| Space Usage | 4 | 14.8% | 0 | 0% | 0 | 0.00% |
| Grouping | 2 | 7.4% | 3 | 9.7% | 12 | 34.29% |
| Long Line | 2 | 7.4% | 5 | 16.1% | 1 | 2.86% |
| **Total** | 27 | 100% | 31 | 100% | 35 | 100% |

In Table 12, the structure reorganize issues are listed. In industrial reviews, *dead code* and *move functionality* were the most frequent issues. In student reviews, these issues were quite evenly distributed. In the refactoring experiment, *long subroutine* and *statement issues* were clearly the most frequent problems. Again, these high frequencies are likely caused by the bias towards such issues in the code that was evaluated in the experiment. In the industrial reviews, *others* included splitting up a large file containing several classes and 2,000 lines of code; issues where a logical piece of code was unnecessarily split into several places; the need to remove wrong couplings between software elements; the need to split up functionality into several implementations; and reducing the number of different error handling mechanisms. In the student reviews, others included having several return statements in a method; using loop variables outside of the loop structure; in-lining a method; beginning indexing at zero instead of one; using negative return values instead of positive; having too many temporary variables; having variables in class scope instead of method scope; having a method with too many parameters; and commented code that should be deleted. In the refactoring experiment, the others consisted of having too many parameters, having too many conditional statements, using direct references to static class variables, using several parameters instead of a parameter object, moving an object creation outside of loop structure, and initializing all variables in the same place.

**Table 12. Structure Reorganize issues**

| Type | Industrial Reviews | | Student Reviews | | Refactoring experiment | |
|---|---|---|---|---|---|---|
| Move Functionality | 17 | 23.3 % | 4 | 4.7% | 5 | 3.3% |
| Long Sub-routine | 9 | 12.3 % | 14 | 16.5% | 77 | 50.3% |
| Dead Code | 21 | 28.8 % | 15 | 17.7% | 0 | 0.0% |
| Duplication | 11 | 15.1 % | 14 | 16.5% | 4 | 2.6% |
| Complex Code | 3 | 4.1 % | 8 | 9.4% | 17 | 11.1% |
| Statement Issues | 2 | 2.7 % | 13 | 15.3% | 39 | 25.5% |
| Consistency | 3 | 4.1 % | 1 | 1.2% | 0 | 0.0% |
| Others | 7 | 9.6 % | 16 | 18.8% | 22 | 14.4% |
| Total | 73 | 100% | 85 | 100% | 153 | 100% |

In Table 13, the structure solution approach issues are presented. In article IV, such issues were all typed under a single issue called *poor algorithm*, but here they are reanalyzed to correspond with article V. To our knowledge, prior work has largely ignored these types of issues.

In all data sets, there were a considerable number of *semantic duplication* cases, which means syntactically different code blocks with equal intent, e.g., different sorting algorithms such as quick sort and heap sort have equal intent but they are not identical at the code level. In the student review and in the refactoring experiment, the semantic duplication mostly referred to situations where certain functionality was redundant or more easily implemented with Java's prebuilt libraries.

*Semantic dead code*, meaning code fragments that when executed have no meaningful purpose and/or have no effect on the result, were only identified in the code reviews. The need to change function call to another function were only identified in the industrial reviews. *Use standard method* means issues in which a standardized way of working should be used. For example, using predefined constants rather than magic numbers; using exceptions for error messaging instead of return values (students working with the Java language only).

The defect type *other* contains a wide range of defects that truly represent the solution approach in its most fruitful form. In the industrial reviews, this type contained implementation changes, such as using arrays instead of other more complex memory management structures; changing the code to enable an easier removal of several data items from the database; using dedicated arrays for each data element instead of a shared array; and using a simpler and more efficient way of keeping records in a database. In the student reviews, other defects included suggesting a simpler way of performing computing and comparison operations, using Java's generics data structures, and caching numeric values rather than re-computing them. In the refactoring experiment, other issues included suggestions to simplifying program logic by preventing improving data structure handling, replacing the use of an "instanceof" operator with method overloading, the use of declarative interface languages such as XUL and avalon instead of Java Swing, and having several search methods instead of a single search method.

*Minor* gathers implementation changes, but the defects were easier to fix and seemed less important than those categorized under other. These defects were only identified in the industrial reviews and in the refactoring experiment. Examples of such issues in the industrial reviews are having a default branch in a switch block, changing an if-else block to a switch-block, and changing comparison element from a class name to a class id. In the refactoring experiment, such issues were using arrays rather than vectors, using while-loop rather than a for-loop, and using Java’s Iterators rather than indexes when accessing variables.

**Table 13. Structure Solution Approach issues**

| Type | Industrial Reviews | | Student Reviews | | Refactoring experiment | |
|---|---|---|---|---|---|---|
| Semantic Duplication | 8 | 10.0% | 7 | 18.0% | 12 | 23.5% |
| Semantic Dead Code | 10 | 12.5% | 4 | 10.3% | 0 | 0% |
| Change Function | 25 | 31.3% | 0 | 0% | 0 | 0% |
| Use Standard Method | 16 | 20.0% | 20 | 51.3% | 2 | 3.9% |
| Create New Functionality | 6 | 7.5% | 0 | 0% | 0 | 0% |
| Others | 9 | 11.3% | 8 | 20.5% | 21 | 41.2% |
| Minor | 6 | 7.5% | 0 | 0% | 16 | 31.4% |
| Total | 80 | 100% | 39 | 100% | 51 | 100% |

## 4.3 Code review defect distributions

In article V, we also studied the distribution of evolvability issues and functional defects found in code reviews. Table 14 shows the distribution of code review findings arranged in three main groups: evolvability issues, functional defects, and false positives. We can see that in both the industrial and student code reviews, over 70% of the findings were evolvability issues. In the industrial reviews, about 20% of the identified defects were functional, while in the student reviews, only 13% were functional. Functional defects existed in the code since module-level functional testing did not reveal all defects. False positives were issues that were identified in the meeting, but that were later or during the meeting found not to be defects after all. If we remove false positives, the share of evolvability issues is 77% in the industrial review and 85% in the student reviews. Thus, based on our data, it seems that roughly 4/5 of the true defects identified in the code reviews were evolvability issues.

**Table 14. Distribution of Code Review Findings**

| Main Group | Industrial Reviews | | Student Reviews | |
|---|---|---|---|---|
| Evolvability issues | 276 | 71.1% | 287 | 77.4% |
| Functional defects | 83 | 21.4% | 49 | 13.2% |
| False positives | 29 | 7.5% | 35 | 9.4% |
| **Total** | 388 | 100% | 371 | 100% |

## 4.4 Human evaluations of software evolvability

So far, this work has presented various evolvability issues found in the source code. Next, we shift our focus from the evolvability issues and look at the aspects affecting human evaluations of the evolvability issues. This is important as it can reveal details about the trustworthiness of the human evaluations and give explanations to the variations in the evaluations. Furthermore, we try to link the widely studied source code metrics and see how well they correlate with human evaluations.

### *4.4.1 Interrater agreement*

We studied the interrater agreement in articles II and III. High interrater agreement is a positive indication of the reliability of the subjective evaluations. Lack of interrater agreement can mean that some evaluators are mistaken in their evaluations or that due to their experience or lack of it, they perform disagreeing evaluations.

In article II, we studied the distributions and standard deviations of the evaluations the company's developers had given about the company's software modules. The evolvability evaluations were performed by answering whether the code smells provided by Fowler and Beck existed in the evaluated modules. We found a case in which the developers had a perfect agreement of the amount of long method smell in a particular module. However, this was unique because for the other smells, the range in seven-point ordinal scale evaluations was three or more for the two modules that had received most evaluations (5 and 6). Further analysis of other modules revealed that the distributions in the smells evaluations were similarly large for the other modules as well. Thus, based on the study, there was at least some disagreement between the evaluators.

However, in article II there was not a sufficient number of data points to use robust statistical methods to analysis the interrater agreement. Thus, we performed another study reported on paper III that tried to assess these issues. Table 15 shows the results of the interrater agreement analysis of paper III. In this study, the evaluators evaluated the existence of three code smells and whether the method should be refactored to make it more evolvable. For the refactoring part, we performed the experiment twice with different evaluators. The interrater agreement is measured by using Kendall's *W*[3]. For those not familiar with this measure, we quote Siegel: "Whereas Spearman's rho and Kendall's Tau[4] express the degree of association between two variables measured in, or transformed to, ranks, W expresses the degree of association among k such variables that are in association between k sets of rankings." Thus, Kendall's *W* can be seen as a correlation between more than two judges. From the measures of interrater agreement in Table 15 we can see that the evaluators had a high agreement on evaluations concerning the long method and long parameter list smells. The agreements concerning the feature envy smell and the refactoring decisions are considerably weaker. However, all the *W* values are significant indicating that the evaluators had at least some level of agreement. The *W* values of the refactoring decision for both experiments are close to each other, which strengthens our results. Based on the study, in article III it appears that the agreement on simple issues is high whereas agreement on more complex issues is lower.

**Table 15. Interrater agreement**

| **Question** | **N** | ***W*** | **p-value** |
|---|---|---|---|
| Exp A – Long Method | 46 | 0.777 | 0.000 |
| Exp A – Long Parameter List | 46 | 0.816 | 0.000 |
| Exp A – Feature Envy | 44 | 0.238 | 0.000 |
| Exp A – Refactoring | 45 | 0.353 | 0.000 |
| Exp B – Refactoring | 36 | 0.397 | 0.000 |

### 4.4.2 *Effect of demographics*

Because people do not always agree on the evolvability evaluations, we wanted to determine if demographics could explain the differences of opinion. The effects of demographics were studied in articles II and III. Results of article II are based on data gathered from a single case company. Results of article III are based on a controlled experiment which tries to verify the results of article II. The differences between the results of articles II and III are also discussed in this section.

[3] Fleiss' kappa is not applicable here as it can be used only for nominal data.

[4] These two are the most recognized ways of measuring non-parametric correlation.

In article II, we studied how the role, experience, and knowledge of developers affected the smell evaluations. We found that regular developers perceived a higher amount of duplicate code than the lead developers. However, lead developers identified more parallel inheritance hierarchies. This result correlates with the idea that regular developers work closer to the code level and that lead developers have more design tasks. When studying how knowledge affected the smell evaluations, we found that developers who reported higher knowledge of the module reported higher levels of lazy class code smell, which refers to a small class that should be removed from the application. Again, this would be expected, since the smells that are difficult to spot require better knowledge of the code.

Also in article II, we also studied the effects of work experience on the smell evaluations by comparing the two developers with the longest work experience in the case company to the rest of the developers. These two developers had worked in the company for seven years, whereas the most experienced of the other developers had been with the company for less than five years; the rest of the informants had worked for the company for three and a half years or less. The two most experienced developers were the only developers who had been working in the company for the entire lifetime of the software products. They had also provided eleven smell evaluations, which made the comparison to the rest of the informants sensible. They tended to observe that the software had many fewer smells, compared to the observations of the other ten developers. A possible interpretation of this result is that the two developers had an emotional attachment to the software since they had written a great deal of it and were reluctant to acknowledge the smells. In addition, the fact that it is easier to understand code and design that you have personally created might affect the evaluations. Thus, it is possible that we actually studied authorship rather than work experience. Another interpretation, suggested by one of the lead developers, is that developers get used to the smells. People who have worked with software products for longer periods of time may understand that complex software products do not always look like textbook examples. Naturally, one should look these results with caution, as the differences between the two experienced developers and the others could be due simply to chance.

We continued to the affect of the demographics in the study reported in article III. This study was performed in more controlled settings. In that study, we collected several demographic variables and used regression analysis to see whether they predicted the refactoring decision or the evaluations of the three code smells: long method, long parameter list, and feature envy. For all cases, we found that demographics were not meaningful predictors, which is somewhat surprising when comparing with the results of article II.

Next, we compare the result of articles II and III. In article II, we studied whether the developer role made any difference in the evaluations. In article III, we did not have any roles since all the subjects were student participating in the experiment. Thus, we cannot make any comparisons. In article II, developers had different amounts of knowledge of the modules in evaluation. In article III, all the subjects had the same knowledge of the software under evaluation. In article III, we asked the students to give self-assessments of their knowledge of the Java programming language, but this variable was not a meaningful predictor. Thus, it could be that general programming language knowledge is not a meaningful predictor, but application-specific knowledge can explain some differences in the evaluations. In article II, we found that the two most experienced developers thought the software evolvability was better than the other developers thought. In article III, we asked the student about their programming work experience, but found that it was not a significant predictor. Thus, it appears that work experience in general cannot explain evolvability evaluations. However, it is possible that in article II the experience was simply a surrogate variable for ownership and emotional attachment that the two most experienced developers had with the software.

Also in article IV, we found some weak evidence of the effects of demographics. We found that more advanced code problems were only identified by two experienced evaluators each. The individual who found both of these problems had 6 years of work experience. Two individuals found only one of these problems and they had 4.5 and 5 years of work experience. Altogether, we had 6 of 36 evaluators who had 4.5 years or more work experience. Thus, it seems that work experience increases the likelihood of an individual detecting advanced problems, but it cannot guarantee it, as only half of the people that had high work experience spotted one of these problems.

### *4.4.3 Code metrics and human evaluations*

We studied the relationship between source code metrics and software evolvability evaluations in articles II, III, and IV. We decided to compare human evaluations and source code metrics because source code metrics have been widely studied in academia, but according to our experience with small- and medium-sized software product companies, they are rarely used in industry.

In article II, we compared developers' code smell evaluations of four smells (large class, long method, long parameter list, and duplicate code) against selected source code metrics. For the large class smell, the developers evaluated that the two modules had no difference when majority of the code metrics suggested that the other module suffered more severely of the large class code smell. However, for the large class smell, the chosen measures were also conflicting. Lines of code, cyclomatic complexity, number of class variables, and number of methods suggested that module A1 had more large class problems, but measure lack of cohesion methods (LCOM) suggested that A2 had more large classes. For the long method and long parameter list, the developers' evaluations correlated quite nicely with the code metrics. For the duplicate code, the metrics and the smell evaluations did not correlate. Based on the data, it seems that the evaluations by all but one developer conflicted with the metrics when it comes to duplicate code. However, in the company copy-paste-modify programming had been used, which resulted in some nearly duplicate code that the tool could not detect. This limitation might have affected the results.

In article III, code metrics were used to create a regression model that tried to predict the students' refactoring decisions and the evaluations of three smells, namely long method, long parameter list, and feature envy. For the long method and long parameter list code smells evaluations, the code metric regression models were able to explain over 70% of the variation. This means that code metrics are good predictors for these two code smells. Closer examination of the regression models revealed that lines of code was the most important predictor for the long method smell, and that number of the parameter was the most important predictor for the long parameter list smell. For the feature envy smell, the code metric regression models were only able to explain 9.8% of the variation meaning that code metrics were not successful predictors of that smell. However, we have to remember that the interrater agreement for the smell was also low as we saw in Section 4.4.1 which can partly explain the results. For the refactoring decision, we had two sets of informants who gave their refactoring decision in two partly different experiments. The code metrics were able to explain 31.9% and 26.1% of the refactoring decision. This means that the used code metrics can partly explain the refactoring decision and they can be somewhat valuable for developers if they are used for suggesting refactoring targets.

In article IV, we studied the refactoring drivers, i.e., evolvability issues, students had identified in the source code, and analyzed whether the identified drivers can be found with tools. Strictly speaking, this analysis has a somewhat larger focus than just code metrics. However, from a practical viewpoint, code metrics and other tools detecting evolvability problems try to solve the same problem, thus they are studied together. We found that many refactoring drivers are suitable for automatic detection. However, some drivers would require new code measures. For example, we are not aware of any previously introduced measures or tools for statement length that would indicate when a single statement should be split into several statements. Some drivers could be partly detected. For example, we can say that a large number of parameters suggest that parameter objects should be used. However, the feasibility of the parameter object must be determined by a human. In addition, we can automatically detect that a method is lacking javadoc comments; however, nothing can be said about the sensibility of the comments. Two important refactoring drivers cannot be automatically detected. Drivers that address the quality of code naming or commenting cannot be detected. Drivers belonging to the solution approach group (called poor algorithm in the article) also cannot be automatically detected.

### *4.4.4 The impact of the evolvability issues*

In the previous section, we saw what types of evolvability issues are detected from the source code. This gives us descriptive information on the evolvability issues, but it does not differentiate the issues based on their importance. For example, it would seem likely that people would consider removing duplication more valuable than finding simple spelling errors in the comments. In study IV, we also studied the connection between the evolvability issues and the likelihood of a positive refactoring decision.

In article IV, we created three different regression models to study how the evolvability issues given by the students affected their refactoring decision. The regression models are shown in Table 16.

**Table 16. Qualitative regression models**

| Model | Adjusted $R^2$ | p-value |
|---|---|---|
| Simple model | 0.692 | 0.000 |
| Target model | 0.403 | 0.000 |
| Pure Drivers model | 0.377 | 0.000 |

First, a *simple model* with two independent variables was created. The first independent variable was the count of evolvability issues given by a student. The second variable was measured on a binary scale and it indicated whether a student had given any positive comments about the method under evaluation. In Table 16 we can see that the simple model has an adjusted $R^2$ of nearly 0.7, meaning that the model is able to explain nearly 70% of the variation in the refactoring decision. The details of the simple model, shown in Table 17, revealed that both independent variables were highly significant and they both had high standardized betas. The variables also have high correlation with each other: Kendall's Tau-b correlation minus 0.613 ($p<0.01$). This indicates that the two variables often act inversely, i.e., if one has high values, the other has low values and vice versa. Not surprisingly, the count of evolvability issues (the negative comments) increased the likelihood of refactoring and the positive decreased it. This indicates that both positive and negative comments affect the refactoring decision.

**Table 17. Predictors in the Simple model**

| Driver group | Std. β | p-value | Correlation |
|---|---|---|---|
| Positive | -0.407 | 0.000 | -0.779 |
| Negative | 0.476 | 0.000 | 0.794 |

The second model, called the *target model,* studied the effect of the evolvability issues grouped into driver groups structure, documentation, and visual representation. The target model is able to explain a little over 40% of the variation, but the decrease compared with The simple model is mostly explained by the fact that the variable measuring the presence of positive comments was not present in this model. The details of the target model, in Table 18, show that structure issues were the most important predictors. Visual representation also had some effect. Documentation did not have any effect on the refactoring decision. It is not surprising that structural evolvability issues are seen as the most important reasons for code evolvability improvement. However, it is surprising that the visual representation have more impact than the documentation issues.

**Table 18. Predictors in the Target model**

| Driver Group | Std. β | p-value | Correlation |
|---|---|---|---|
| Structure | 0.610 | 0.000 | 0.608 |
| Visual Representation | 0.197 | 0.000 | 0.199 |
| Documentation | 0.035 | 0.400 | 0.016 |

The third model, called the *pure drivers model*, contains only the pure evolvability issues (referred as pure drivers in article IV) and excludes the positive comments. This model aimed at revealing the individual issues that had the greatest effect on the refactoring decision. The model had adjusted $R^2$ of only 0.377 indicating that only 37.7% of the variation is explained by the model. The model has a $R^2$ (non-adjusted) of 0.587, but the model suffers when the adjusted $R^2$ is calculated, because it has a high number of independent variables (61 drivers). Next, we present the most significant drivers. The three strongest drivers were *long method or extract method* (std. beta 0.481), *not enough blank lines* (std. beta 0.237), and *long statement* (std. beta 0.210). Especially the long method or extract method driver appears to be a strong indicator of the refactoring decision. The problem with the pure drivers model is that there were more than 60 predictors and many of the predictors were seldom mentioned by the evaluators meaning that the data matrix is sparse. More than 40 predictors are mentioned less than ten times. Furthermore, a regression model that was equally good with the pure drivers model was created without the drivers that were mentioned less than five times. The problem with the pure drivers model is that there really is not enough data to create a good regression model for more than 60 predictors.

This section has presented different regression models measuring the impact of the various refactoring decisions. The regression models indicate only the majority opinions and big trends. They do not give any indication of the real effect the evolvability issues have on future development effort. The models may also ignore important evolvability issues that are discovered by only the more talented developers. Regardless of these limitations, they are valuable as they shed at least some light on what are seen as important evolvability issues and what are not.

# 5. DISCUSSION

First, this chapter discusses the results and answers the research questions. With each research question, this chapter discusses the meaning of the results, their generalizability and practical utility, and performs comparisons to related works. Finally, the limitations of this work are discussed.

## 5.1 Software evolvability issues

### *5.1.1 RQ 1.1*

*How can the evolvability issues, either presented in the literature or identified by humans, be classified?*

This research question was studied in Section 4.1 where various classifications were presented. Section 4.1.4 presented the classification that combined the classifications of code smells and the classification of the evolvability issues identified by humans.

We see that the generalizability of the classification is affected by two factors. First, the data sets that were used for creating the classification might affect the generalizability. For example if our data sets are very different from the usual case, then our classification might not be generalizable. Second, the researcher bias might affect the created classification and thus it may lead not generalizable classifications. To assess the generalizability of the classification created mainly by one researcher based on three data sets is difficult. Thus, only time will tell how generalizable the classification will be. However, we may speculate that top-level classes of the evolvability issues classification, documentation, visual representation, and structure are even generalizable beyond current programming languages. For example, we might be able to distinguish the same types of problems from the specifications of traditional engineering, such as shipbuilding and building design. In traditional engineering, documentation would refer to the clarity of the writing and chosen terminology, visual representation would refer to the layout of the document and the clarity of the plan part drawings, and structure would refer to the organization of the specification and the chosen engineering solutions. Naturally, this is only a speculation, as we have no expertise in traditional engineering specifications.

Next, we try to assess the classification's usefulness. As previously noted in Section 1.2.1, we hope that our classification will serve four purposes. First, the classification increases the body of knowledge in software engineering and increases the understanding of the nature of software evolvability issues. Second, the classification can be useful when creating company coding standards, code review checklists, and assigning roles to the participants of code reviews. Third, the classification can be used as a basis for evolvability assessments (see, e.g., [1]). Fourth, the defect classification can provide input for creating automated defect detectors or developing new programming languages. At this point, it is difficult to say how well the classification achieves these goals and only time will tell.

Next, we compare our classification with the ones presented in prior studies. We are only aware of one academic study in which evolvability issues gathered from empirical data would have been classified [113]. Several works have presented evolvability issues that should be avoided, but they are based more on personal experience than data. For examples of such studies, see Section 2.1.1. Siy and Votta proposed the classes of documentation, style, portability, and safety. A high-level comparison reveals that both classifications contain class documentation, but they are not equivalent. Our definition of documentation is wider since we considered the naming of code elements (e.g., variables and routines) to be a part of documentation, whereas Siy and Votta only considered the code comments. Furthermore, we consider documentation that is supported by the programming language (e.g., key word “final” in Java) to be part of the documentation group and these types of issues were also left out from the documentation class defined by Siy and Votta. Our classification does not have anything that would be comparable with portability and safety, as we would classify such issues as functional defects rather than evolvability issues. The style class consists of evolvability issues that in our classification belong to structure, visual representation, and documentation. The style class by Siy and Votta have a subgroup called clean-up whose description is very similar to the structure group of this work. Similarly, style issues have a subgroup called cosmetic that is very similar to our group visual representation. Style issues also have a subgroup called renaming that in our grouping belongs under documentation. Finally, the style issues had a subgroup called debugging that meant adding debugging statements to the source code. In our work, such issues were seldom found and they were placed under the documentation group. To conclude, we can say that many similar low-level evolvability issue types can be found from both classifications. However, there are also significant differences on how the classifications are organized.

Additionally, there have been many works classifying software defects that have also considered evolvability issues [11, 17, 32, 45, 54, 63, 68]. However, in those works, only Grady [54] has included a class for structural issues, called *module design*. We see the lack of structural issues in those classifications as a major shortcoming; such issues are very common issues in many source code documents. Naturally, there are three possible causes: first, structural issues were simply ignored; second, they were classified under some other defect class; or third, such defects did not exist in the code. As the last explanation seems very unlikely, the results of the studies are most likely somewhat biased when it comes to the classification of evolvability issues.

### *5.1.2 RQ 1.2*

*What types of evolvability issues are identified in the source code by humans and how are they distributed to different evolvability issues?*

This research question was studied in Section 4.2 where evolvability issue types and distributions were presented. Additionally, the classification that was created as the answer to RQ1.1 contributes to this research question as it identified the high-level evolvability issue types. Next, we discuss the evolvability issues under the three main categories: documentation, visual representation, and structure.

#### *5.1.2.1 Documentation*

Documentation is information in the source code that communicates the intent of the code to humans. We separated documentation into two subgroups: textual and supported by language. Several past studies have recognized documentation as an issue that should be fixed; for example, Siy and Votta [113], El Emam and Wieczorek [45], and Chillarege [32] report documentation defects in the code review context. However, in the past studies [32, 45, 113], documentation consisted only of code commenting. The past studies have not considered supported by language issues to be part of documentation. Furthermore, code element naming is also excluded from code documentation, which is surprising since clever code element naming can significantly reduce the needed code comments. Based on this comparison, our documentation group is larger than what has been presented in prior works.

In our studies, documentation had a 35% share of evolvability issues in the industrial code review, a 46% share in the student reviews, and a 25% share in the refactoring experiment.

We were able to calculate proportions of textual documentation defects (naming and commenting) from past studies [32, 45, 113] This comparison shows that textual documentation had a 17% to 50% share of all code review defects. Therefore, it seems that a considerable share of evolvability issues is related to documentation, but the actual shares fluctuate considerable between different contexts.

Our data shows that textual issues were more frequent than the supported by language issues. The supported by language issues had the highest share in the industrial settings with 1:2 ratios against the textual issues, second highest in the student reviews with 1:5 ratios, and lowest in the refactoring experiment with 1:30.5 ratios. Thus, the majority of documentation issues come from the textual documentation, i.e., good naming and commenting, which have long been considered part of a good programming style.

Further analysis of textual documentation shows no consistencies in the ratios between commenting and naming. It appeared that in our study the industrial developers were keener to improve naming while the students found more commenting issues. In prior work [45], when studying defect types identified in two code review sessions, naming issues were seldom identified. In the first review of that study, there were no naming issues present when half of the code review defects identified were classification as commenting defects. In the second review, the ratio between commenting and naming were 17:1. On the other hand, study of the Eclipse IDE usage in [93] shows that renaming was used by all of the 41 developers participating in the study, and it was the most used refactoring feature. Similar numbers can be found in [125] which indicates that renaming is very common practice when compared with other refactorings. We must note that past studies referred in this section are heterogeneous; therefore, comparing them is likely to lead to somewhat conflicting results. Nevertheless, it seems very strange that El Emam and Wieczorek [45] found hardly any naming issues while Murphy et al., and Xing and Stroulia [93, 125] found naming changes to be one of the most frequent changes in the source code. One possible explanation is that people can consider different fixes for the same problem. For example, if code is unclear, one individual might suggest commenting while another might suggest additional informative code element naming. Another explanation could be the applied organizational standards. For example, if an organization has a tight policy about code commenting, it will naturally increase the code commenting issues that are found and fixed.

As previously mentioned, prior studies did not consider supported by language issues to be part of the documentation category. Nevertheless, we found empirical data of such issues from a study by Xing and Stroulia [125]. From their study of the evolution of Eclipse IDE, we found that through version 2.0 to 3.1, there had been 4,442[5] changes that we consider fixes for supported by language issues. They also found 4,582 renamings indicating that a proportion of supported by language issues could be equal to program renaming. They also reported individual numbers between three versions (from 2.0 to 2.1, from 2.1.3 to 3.0, and from 3.0.2 to 3.1) and in each of these, the proportions between renamings and supported by language issues were about equal. In our study, the ratios between renamings and supported by language issues were roughly 3:2, 3:2, and 27:2. The last ratio came from the refactoring experiment in which hardly any students reported supported by language issues. Thus, it seems that in our study, we had a somewhat higher proportion of renamings. However, it is difficult to make a comparison as in the study by Xing and Stroulia in which the data was automatically collected from the version control. Thus, we do not know which of their

[5] This number is calculated from the article’s Table 3 by combining visibility changes, data type changes, and non-access modifier changes.

changes were actually due to evolvability issues and which were changes due to the need for new features in the program, e.g., the changes could have been made to meet the needs of new features rather than evolvability issues.

Our studies and the prior works show that there is plenty of empirical evidence of the existence of documentation issues and that the proportion of documentation issues also appears to be significant when compared with other evolvability problems. We also believe these results are generalizable to many software development contexts. Our study did not assess the cost related to evolvability issues. However, prior works [53, 120] have shown that proper documentation improves program comprehension and it has a larger impact than code structure.

#### *5.1.2.2 Visual Representation*

Visual representation means defects that hinder program readability for the human eye. Many of the past studies have not mentioned visual representation issues. It could be that such issues have been seen as too trivial. However, we must stress that there is nothing new about visual representation issues since such issues have long been discussed in the coding standards, such as Sun Microsystems' Java Coding standard [118]. Siy and Votta categorized visual representation as a cosmetic style issue [113].

In our data sets, visual representation issues had the lowest proportion of defects with percentages varying from 9.8% to 14.7%. In the study by Siy and Votta, visual representation had a share of 11%. Thus, based on the data sets, it seems that visual representation has a consistent 10% share of evolvability issues. Naturally, as we only have four sources, this could be due to coincidence. Nevertheless, it is safe to say that proportion wise, visual representation issues represent a minority of the source code evolvability issues. This information is important since it indicates that evolvability issues are not mostly simple layout issues that could be easily fixed with automatic pretty printers. Nevertheless, we need to point out that visual representation is important since it has been shown to have an effect on program comprehension [88] and debugging efforts [53].

#### *5.1.2.3 Structure*

Structure is clearly distinguishable from documentation and visual representation because the latter two have no impact on the program runtime operations or the syntax tree generated from the source code. Prior works on software evolvability have listed various issues concerning source code structure as can be seen in Section 2.1.1. Perhaps one of the most important findings in our study is the distinction between two of the structural evolvability issue types: solution approach and re-organize. Re-organize issues have been widely recognized and studied in the past. Fixing the re-organize issues has been studied extensively under the term of refactoring [87]. Furthermore, the automatic detection of re-organization issues has been studied through static code analyzers and code metrics tools. Prior works in the software engineering domain have not studied or discussed solution approach issues, although some of the evolvability issues in the past studies can be classified as solution approach issues. Formally, recognition of the solution approach issues highlights the fact that software development is creative work-based on skill, knowledge, experience, and education, i.e., a craft that cannot be completely controlled with automated and statistical approaches that have been successful in the manufacturing industry. We believe that one phenomenon that is likely to increase solution approach issues in the future is the growth of software libraries and existing solutions. In article V, several solutions in which the modification used prebuilt libraries instead of self-programmed solutions were used. If we think back to the

1970s when many software evolvability criteria were first created, very few, if any, code libraries existed.

The proportions of organization defects have been previously reported by Siy and Votta (under the name clean-up), but comparison of the defect proportions reveals no consistencies, as they vary from 26% to 55%. The refactoring study by Xing and Stroulia [125] reported numbers of organization move functionality issues (in their work called *move entity*). However, since their study did not report all evolvability issues, we need to compare their numbers with their shares of program renamings instead of the proportion of all evolvability issues. In their study, move functionality and renamings had a ratio of roughly 1:2 when we had 1:3, 2:15, 1:5. However, as previously noted, we cannot be sure which of their move functionality changes were evolvability issues and which were caused by the need for new features.

The effect of structural issues have been studied widely in the past. For example, [9, 40, 77, 107] use code metrics to measure code structure and show that poor structure results in increased development efforts. Thus, there are practical benefits of understanding and fixing structural issues.

### *5.1.3 RQ 1.3*

The results for research question 1.3, "*What is the distribution of evolvability issues and functional defects found in code reviews,*" were presented in Section 4.3.

Based on our study, roughly 75% of the findings identified in the code reviews were evolvability issues that would not cause runtime failures. This research question was confirmatory as Siy and Votta [113] had previously proposed: based on data from a single company, most code review findings are evolvability issues. Our results confirm their findings.

Comparing our results with the numbers of Siy and Votta shows that our study has a slightly higher proportion of evolvability issues, a slightly lower proportion of false positives, and similar proportion of functional defects (see Table 19). Siy and Votta used the defect data from the repair form after the author had made the fixes. We observed the defects encountered during the code review meeting in the industrial reviews and used the defect logs returned after the meeting in the student reviews. We used *discovered* defects while Siy and Votta used *fixed* defects. Therefore, we can speculate that had we observed the actual fixed defects, the number of false positives may have increased and the number of evolvability issues may have decreased, as the author might have disregarded some defects, considering them false positives.

We recognize that quality assurance made prior to the code review can have a significant impact on the results. For example, studies that have used uncompiled code have found large amounts of syntax errors. In our case, developers performed automatic unit testing or quick functional testing prior to code review. We think that this is the most realistic scenario for code reviews. Unfortunately, Siy and Votta [113] did not reveal what quality assurance activities were performed prior to code review in their study.

**Table 19. Comparing Our Results with the Results of Siy and Votta**

| | Industrial Reviews | Student Reviews | Siy and Votta |
|---|---:|---:|---:|
| Evolvability issues | 71.1% | 77.4% | 60% |
| Functional defects | 21.4% | 13.2% | 18% |
| False positives | 7.5% | 9.4% | 22% |

To determine whether the large number of evolvability issues (compared with functional defects) in our study and that of Siy and Votta was simply due to chance, we attempted to use public data available from prior code review studies. Unfortunately, we only found three studies [32, 45, 101] with a sufficient number of defects and enough information to try to perform an approximation of the defect distributions. We excluded the PSP data sets of Humphrey [61] and Runeson and Wohlin [108], as they had high shares of syntactical errors because they performed the reviews before compilation.

Figure 7 compares eight data sets, showing the proportions of functional defects and evolvability issues. In five of the eight data sets, the majority of the defects are evolvability issues, and in seven of the data sets, over half of the defects are evolvability issues. Based on the figure it seems likely that the majority of the defects detected in code reviews are in fact evolvability issues. However, there is considerable variation between the studies, for which we think there might be three explanations. First, quality assurance performed prior to code reviews affects the number and types of defects detected. Unfortunately, such information is not available in studies Si, O, E1, E2, and C (see Figure 7 and caption for detailed references). In study So [114], a set of acceptance tests provided by the course staff was used to ensure a minimal level of functionality. Only minimal functionality was tested, as the purpose of the So study was to compare different functional defect detection methods. In our studies, the code authors had personally tested the application and were either personally responsible (industry case) or had been graded earlier based on the number of faults (student case). Thus, it is likely that the higher number of functional defects detected in reviews in So is explained by quality assurance performed prior to code review. Second, misclassified defects or mistakes in our reclassification in studies E1, E2, and C can cause part of the variation. In those studies, the authors had no class for structural evolvability issues, and as previously discussed in Section 2.3, it is possible that such evolvability issues were ignored or categorized as functional defects. Furthermore, in study C, there appeared to be a defect class that possibly contained both functional defects and evolvability issues. Additionally, in those studies, the authors did not make a distinction between functional defects and evolvability issues. Therefore, we made the separation into evolvability and functional defects based on the defect type description, and it is possible that we have misclassified some classes. Thus, it is possible that in those studies, the proportions of evolvability issues would have been higher, had the original data been available. Third, other unknown context factors can also explain the variation, for example, applied coding standards, the strictness of the code review process, and the company culture. Because of these shortcomings, one should study Figure 7 with some caution.

To summarize, we have three data sets in which the authors compared the proportions of functional defects and evolvability issues, and these data sets show that the ratios (evolvability issues to functional defects) fall between 5:1 and 3:1. In addition, four other data sets that are more unreliable produce mixed results.

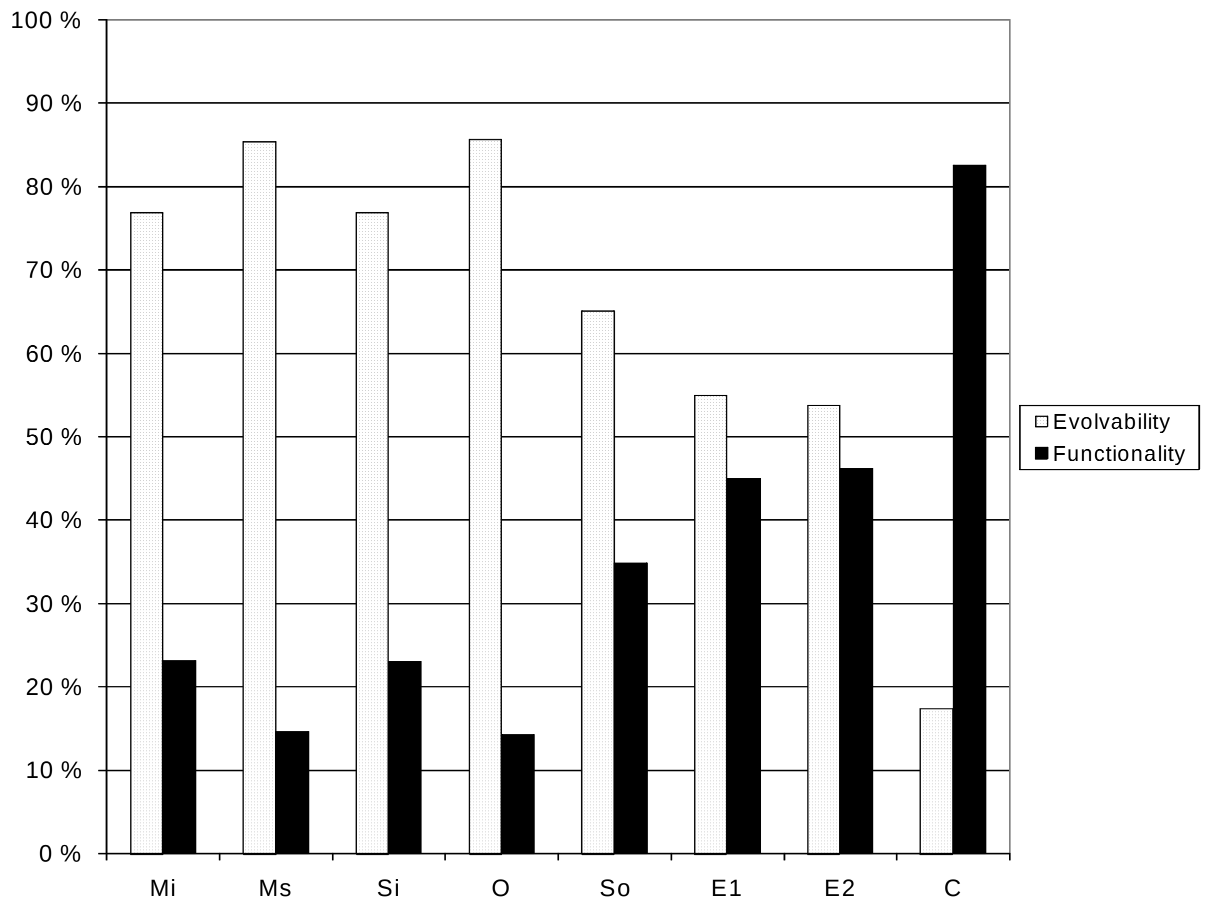

**Figure 7. Defect distribution between functional defects and evolvability issues**
**Mi is industrial data, Ms is student data, Si** [113], **O** [101], **So** [114], **E1 and E2** [45], **and C** [27, 32]

## 5.2 Human evaluations of software evolvability

### *5.2.1 RQ 2.1*

Regarding research question 2.1, "*Do humans achieve interrater agreement when performing code evolvability evaluations*," we found high interrater agreement for simple evolvability issues such as method length. We found lower interrater agreement for more complex issues, such as code refactoring. It seems natural that agreement is higher on the simple issues than it is on the ones that are more complex. However, it is too early to speculate whether this result is generalizable and holds in other contexts.

Our results indicate that controlling and preventing simple evolvability issues can be achieved by coding rules and motivating developers to follow them. To control simple evolvability issues, it is not necessary to have senior developers or automatic metric tool detectors check the code. Naturally, this is based on the assumption that the developers are trustworthy. However, when it comes to more difficult evolvability issues, ranging from coupling and class responsibilities heuristics to highly domain specific implementation rules, it is a good idea to have some quality control measures in place. For the more complex evolvability issues, it is possible that not all developers understand them or that some developers have different interpretations of them.

In a study by Shneiderman et al. (pp. 134-138 [112]) (see Section 2.2 for more details), the results showed that in half of the evaluations, three out of four evaluators agreed on the evaluations (answers differed by one at most). However, in only 43.1% of the evaluations, the range between all the evaluators was two or less. Thus, it seems that in the Shneiderman study, there also existed agreement and disagreement in the evaluations. Shneiderman's book contains partial data sets that we used for some calculations (unfortunately, Kendall's *W* could not be calculated because evaluators' id could not be traced through the programs that they had evaluated.). The three questions with the best agreement considered the chosen algorithm, program comprehensibility, and modifiability. The lowest agreement was on the four questions addressing compilation and machine independence, whether proper code layout was used, and for the quality of high level program design. Shneiderman's questions were more general and to some extent more subjective than ours, which limits the comparability of the results. In Shneiderman's questions, it is difficult to say which evolvability issues are simple and which are complex, thus, the data does not support or refute the idea that there is higher agreement for simpler issues.

Shepperd [111] collected the subjective opinions of the maintainers for 89 modules of aerospace software that totaled around 30,000 lines of code. Each developer on the maintenance team was asked individually to classify each module on an ordinal scale, from one to four, on the perceived difficulty of a hypothetical maintenance task. For 73% of the modules, the range was one or less, and thus the researchers concluded that there was a strong correspondence between the individual evaluations. However, as no detailed data is given, it is difficult to assess the study in more detail.

To summarize, comparison to prior work [67, 70, 111, 112, 123] is challenging due to the lack of proper representation of the evaluation data [100, 111, 123], lack of statistical power [70, 82], and use of nonstandard statistical methods[6] [82, 111, 112]. All prior work lacks calculation of statistical significance on the interrater agreement and Kendall's *W* making it hard to tell whether the raters really agreed on the evaluated software.

### *5.2.2 RQ 2.2*

Research question 2.2 is "*Do the demographics of humans affect or explain the evolvability evaluations, and if so, how?*"

In article II, it appeared that the demographics might have some impact on evolvability evaluations. In that study, we found that regular developers thought there were more code level issues where the lead developer thought that there were more high-level evolvability issues. We also found that developers with better knowledge of the system thought there were more issues that were difficult to find based on superficial examination. Unfortunately, in article II, we had only a limited number of developers and could not utilize robust statistical methods. In article III, we further investigated the effects of demographics. However, in that study, we found that in laboratory settings using statistical methods, demographics do not affect the evolvability evaluations. The differences between the studies in articles II and III were discussed in Section 4.4.2. It is clear that in article III, the laboratory settings and the lack of personal attachment and ownership to the code that were present in article II affected the results. In article III, we had no differences in evaluators' roles, knowledge, and ownership and thus could not study them. To answer this research question, we can conclude

[6] The researchers have calculated the percentage of answers that were off by *n* steps in their ordinal scale, or they calculated averages and standard deviations from the ordinal scale.

that people’s relationships to the code may affect the evaluation results, but people’s general backgrounds, e.g., experience and education, do not affect the evaluation results. The results, assuming that further studies can confirm them, can be used when assessing the reliability of human code evaluations. In other words, one should be cautious when getting evaluations from developers who have been heavily involved in the development. Of course, this is not new since independent evaluations are often used when reliable assessment is needed. Further, in our circumstances, the generalizability of the results is not high. Thus, one should realize that the answer for this research question is only partly conclusive, and the result should be used with caution.

In related work, we are not aware of any studies where the effect of demographics would have been studied in relation to the evaluations of code evolvability. If we broaden the scope, we found some studies that researched the effect of demographics in the field of software engineering. There is work on the differences between novice and expert coders that has focused on the cognitive process and improving novices’ performances, e.g., [65, 126]. A study of programmer performance found that better academic record and higher age contributed positively to performance [41]. There have also been many studies of personal factors and programming aptitude (see [104] for details). Although such related work exists, it is too far removed for any meaningful comparison with our work.

### *5.2.3 RQ 2.3*

Research question 2.3 is “*What is the relationship between evolvability evaluations and source code metrics; do the evaluations and metrics correlate or explain each other?*”

In article II, we found that code metrics and smell evaluations were somewhat conflicting in large class and duplicate code evaluations. There was agreement between smell evaluations and code metrics in the long parameter list evaluations. In article II, the developers performed the evaluation on the modules they had primarily worked with, but the answers were most likely based on their recollections. In addition, in article II, the number of developer evaluations was too small for statistical analysis.

To fix these shortcomings, article III uses a high number of student subjects in a controlled experiment. In that case, we found that simple evolvability issues, long method and long parameter list, were highly correlated with the appropriate source code metrics. When studying the code metrics, the refactoring decision, and the more complex feature envy issue we found that correlation was lower. Comparing these results with the results of research question 3 reveals that both the interrater agreement and the metric regression models have a similar two-fold structure. Both perform well on simple evolvability issues evaluations. Similarly, both the interrater agreement and the regression models have low values when it comes to the refactoring decisions and feature envy issue evaluations. There is a connection between these two, and it is caused by the fact that the source code metrics of any method will remain the same even if there is disagreement between raters. Thus, if there is disagreement about whether a certain method should be refactored, it automatically means that the code metrics of that method cannot make up a strong regression model that would predict the refactoring decision.

In article IV, we continued our study on the relationship between evolvability issues and code metrics. In article IV, we studied the evolvability issues humans find in the source code and found that only part of these can be found with automatic tools. Some issues can be partly detected, for example, if a human suggest that a parameter object should be used we may be able to detect a large number of parameters with metrics tools. Other issues cannot be detected at all. For example, issues related to code element naming or code commenting may not be detected. In addition, issues in which a better solution approach is suggested are nearly impossible to detect with tools, e.g., instead of having this functionality implemented in a code, functionality in the code library should be used.

The results from all three articles seem to have one common theme. The simpler and easier to detect the evolvability issue is for both humans and the tools, the higher the correlation is between code metrics and human evaluations. For example, duplicate code is a simple issue but its detection can be hard both for human and tools. Humans have difficulty finding duplicate code from a large code base and tools have problems when duplicated code is slightly modified in the duplications. An example of a simple and easy to detect issue for both is method length and thus there should be high correlation in human and machine evaluations. It is difficult to assess the generalizability of this finding, but since the same phenomenon was present in all three studies, it seems plausible to believe that it would be generalizable.

A practical implication of the study is that humans cannot be replaced with tool-based evolvability analysis. However, one should use tools to detect issues that are suitable for them as this leaves humans more time for productive work. Thus, companies should make sure that prior to code review, the code has passed a static analysis detector, so that no time is wasted on finding the issues that could be found efficiently with tools. Naturally, there are a number of practicalities that may prevent such a scenario, e.g., it would require a combination of several tools for finding all the issues we want to find automatically or it would take too much effort to get the tools. However, all those excuses should be compared against the gained benefits; it might be that it is not worthwhile to find all issues automatically.

In related work, we have found several studies in which the relationship tool-based source code metrics were compared against human evaluations of software maintainability. Kafura and Reddy [67] studied the relationship between software complexity metrics and software maintainability. Maintainability was measured using subjective evaluations by system experts. They concluded that the expert evaluations on evolvability were in conformance with the complexity source code metric. Shepperd [111] validated the usefulness of information flow metrics on software maintainability and found that correlation between the subjective maintainability evaluations and information flow metric was high (0.7). Kataoka et al. [70] studied the usefulness of improving the software quality with refactoring and reported on a comparison between human evaluation and software metrics. According to the researchers, the subjective evaluation of an expert on the effectiveness of refactorings correlated *quite well* with the improvement in coupling metrics. Genero et al. [52] studied the maintainability of UML-class diagrams. The researchers showed that subjective evaluation of understandability, analyzability, and modifiability of UML-diagrams correlated with various class level metrics. Oman et al. [100, 123] used subjective evaluations to create a metrics-based maintainability measure. It is, therefore, quite natural that their metrics correlated well with subjective evaluations. Recently, Anda [2] compared expert judgment and structural measures in assessing software maintainability. The research found that the expert opinions and software metrics mostly corresponded to each. However, the research suggested that several issues of software maintainability were not captured by source code metrics. All the related works have studied code metrics in relation to a higher-level concept of maintainability and have found that maintainability evaluations have been at least somewhat in correlation with the chosen metrics.

On the other hand, we have studied the evaluations of more concrete issues, such as method length, against the respective metrics. Our human evaluations of the refactoring need are comparable to the maintainability evaluation of the prior works. We have found that the correlations between evaluations and metrics are higher for the simple issues and are lower for the more complex ones. We have also showed that there are evolvability issues that cannot be measured or automatically detected. Thus, it seems awkward that prior studies have tried and been successful in correlating maintainability with each metric that was chosen in a particular study. Our results seem to be somewhat conflicting to prior works since based on our studies, it seems that the concept of maintainability cannot be well correlated with any single metric and even when using several metrics to create regression models, there is still a lot of unexplained variation in the human evaluations.

#### *5.2.4 RQ 2.4*

Research question 2.4 is "What evolvability issues are seen as the most significant by human evaluators?"

This was studied in article IV where we considered the evolvability issues found in the source code and linked them to the refactoring decision. We found that both positive and negative comments affect the refactoring decision, meaning that the count of evolvability issues (the negative comments) increased the likelihood of refactoring and the positive decreased the likelihood of refactoring. In practice, this means that if a method has some evolvability issues it can decrease the likelihood of becoming a refactoring target with having some positive aspects, e.g., good commenting to compensate long and complex structure. In the article, we also studied the effect of the different types of evolvability issues. We found that structure was the most important predictor refactoring decision. Visual representation had a much smaller effect and somewhat surprisingly, documentation had almost no effect on the refactoring decision. We also studied the detailed evolvability issue types and found that the need to perform extract method refactoring, i.e., long method, was the most important individual predictor of a refactoring decision.

Based on the results, we can say that, at the method level, refactoring decisions are affected by both positive and negative aspects of the code, structural issues are seen as the most important groups, and method length is the most important individual issue. However, the generalizability of the results is low as we had limitations: student subjects, a small application, and only ten methods were evaluated. It is possible that in different, perhaps more realistic settings, the results might have differed, e.g., in our settings, we only had little duplicated code, thus, the issue was not an important predictor of the refactoring decision according to the regression models.

Currently, we are not aware of other studies in which qualitative answers concerning code evolvability would have been linked to code refactoring decisions or to evolvability evaluations. It could be that previous researchers have found this topic, the significance of various evolvability issues, too subjective and context dependent.

## 5.3 Limitations

The individual articles of this study have discussed the limitations of each study. Therefore, this section presents only a summary of the most notable limitations.

### *5.3.1 Software evolvability issues*

Next, we address the limitations related to our research on research question 1.1. We created a classification of software evolvability issues. Thus, we need to assess its validity and the process that was used to create such classification. In article V, we validated the repeatability of the created classification. The agreement between two classifiers can measured using Cohen's Kappa [35]. According to El Emam and Wieczorek Kappa values above 0.78 indicated excellent agreement. For the evolvablity issue classification presented in this thesis, we achieved a Kappa of 0.79. Admittedly, there still are borderline cases for which it is difficult to say whether an evolvability issue belongs to one group or the other. In article I, we found weak empirical evidence supporting our code smell classification when we found that there was more correlation within the classification groups than there was between the groups. In articles IV and V, we used qualitative research methods to analyze the evolvability issues and created the classification groups based on this analysis of the empirical data. However, it is likely that researcher bias has affected the result as the entire analysis was conducted by the author of this dissertation. In articles IV and V, we used data from three completely different sources, thus, it seems unlikely that the data source could have biased the results. In article I, the creation of the classification was based only on analytical analysis of the list and description code smells provided in the literature. Thus, it is possible that classification of article I does not really reflect real life phenomenon, even though the list of evolvability issues was provided by highly regarded industry professionals. Currently, we are not able to provide answers of the external validity and applicability of the classification. Only time will determine whether this classification will be considered useful by the software engineering community.

Next, we address the research question 1.2, the types of identified evolvability issues and their distributions. The issues identified in the source code and their distributions to different classes are affected by many factors. We only had source code from three sources, although we had many samples of source code that were evaluated in the last two. With three sources, we can provide only preliminary evidence of the types and distributions because the evolvability issue types and distributions may fluctuate heavily between different sources. In the first research question, the researcher bias may have affected the results; a researcher with a different background may have recognized different defect types and classified the defects differently.

Research question 1.3, studying the share of functional defect and evolvability issues found in code review also had limitations. The most notable being that we studied the code review defect distributions of code that had already passed lightweight functional testing. This naturally decreases the amount of functional testing. However, we think such a scenario represents a realistic view, since, in our experience, code reviews are often performed after initial testing. It would seem awkward to perform team-level code review before the creation of unit tests when there are methodologies advocating that unit tests should be created even before the code is written [15].

### *5.3.2 Human evaluations of software evolvability*

In studying research question 2.1 and 2.2, interrater agreement on the evolvability evaluations and the affect demographics have on evolvability evaluations, we had two data sets in article II and III. Both data sets had limitations. Limitations in article II were due to using a Web survey as a data collection method, relying on developers' recollections in making the evaluations, and having a limited number of subjects. In article III, all of the issues of article II were fixed, but we had a new set of limitations due to using student subjects in laboratory type experiments. Thus, in article III, we could not study the effect of code ownership on evolvability evaluations that was studied in article II.

Research question 2.3, the relationship between evolvability evaluations and source code metrics, was studied in articles II, III, and IV. The limitations of research questions 2.1 and 2.2 also apply to this research question. In addition, in article two we could not measure all types of evolvability issues in all the modules, which reduced the number of data points even further. In article III, where we linked the source code metrics to evolvability evaluation through regression models, the subject only evaluated ten different methods. This reduced the variability in the evaluated methods and most likely created a bias in the regression models. With, for example, 100 different evaluated methods, the regression models would have had a larger and more variable set of methods that would have produced different metric values for the dependent variables. In article IV, we analyzed whether each type of evolvability problem could be detected with source code metrics. This type of approach naturally has limitations, as it is not based on empirical data or even on the opinion of an expert board. Rather, it represents the author's attempt to point out what types of issues cannot be detected with tools.

Research question 2.4, the significance of the evolvability issues, was studied in article IV. The answer to this research also has limitations as it is based only on a single data set. It is possible that an experiment with a different set of methods and people would have resulted in different results. For example, had the evaluated code contained a lot of duplication then it is likely that the duplicate code would have found as a very significant predictor of the refactoring decision. Thus, due to the limitations, the answer to this research question should only be considered as a hypothesis for further work.

## 5.4 Practitioner's implications

This section summarizes the most important findings of this study and the literature from a practitioner's point of view and gives suggestions for how the research results could be applied in practice.

- Based on the literature, software evolvability explains 25% to 38% of the costs of software evolution.
- An established and empirically grounded defect classification (consisting of both evolvability and functional defect types) should be used as a baseline when building checklists for code reviews or when creating coding standards for the developers, to ensure that most typical defect classes are covered.
- Code reviews seem particularly valuable for software product or service businesses in which the same software or service is modified and extended over the years. However, organizations working with customer-specific projects may elect to skip code reviews, as the higher cost of subsequent evolution is often paid for by the customer organization. Naturally, the firms in product or service businesses should consider targeting reviews for those modules that are likely to be modified in the future.
- Ninety percent of the evolvability issues belong to structure and documentation groups that cannot mostly be automatically detected and fixed. Thus, a human element is still required in the process of detecting and fixing evolvability issues.
- When studying structural evolvability issues, one should consider alternative approaches rather than trying to work from the existing solution. Specifically, one

should think whether there is functionality that is already implemented elsewhere. Our research found many cases where functionality existing in the code library had been reimplemented in the source code, as the author was unaware of the existence of functionality.

- Tools can help detect simple structural defects and the code metrics and subjective human evaluations correlate well for simple issues. Organizations should also consider using pretty printers for automatically fixing visual representation defects. Having tool help for simple issues would allow people to focus their valuable time on issues that are more complex.
- Humans have good agreement on simple evolvability issues, but they can have disagreements on more complex evolvability issues. Similarly, disagreement may arise whether a certain piece of code needs refactoring. Thus, developers should collaborate on the decisions that are more complex.
- Human evolvability evaluations are affected by code ownership and organizational role. However, factors such as education and experience have little impact. Thus, the code author’s assessment of the code evolvability should be studied with caution regardless of his or her experience and knowledge.

# 6. CONCLUSIONS

## 6.1 Contributions of the research

This research has studied software evolvability issues at source code level. We make four contributions.

First, we have provided classifications of the evolvability issues based on analysis and empirical data. We hope that such classifications facilitate understanding about the evolvability issues. For example, universities can use it when teaching evolvability issues and perhaps companies can organize coding standards, code review checklists or code review roles based on it. Article V has more detailed description of using defect types when creating roles for code reviews in the context of scenario based reviews and defect based reading by Porter et al. [105]. Our classification consists of many issues that have also been recognized in prior works. However, based on empirical data we also recognized a new evolvability issue type called solution approach, which indicates the need to rethink the current solution rather than reorganize it. For solution approach issues, we are not aware of any research presenting or discussing such defects in the software engineering domain. However, we understand that such issues have always been present in software engineering but they have not been formally recognized in the literature.

Second, we provide empirical results of the types and distributions of the evolvability issues. We found interesting evolvability issue types, such as semantic duplication and semantic dead code, which were not widely discussed in prior works. We found that visual representation accounts for a very small share of evolvability issues. In all cases, it was less than 15% of the evolvability issues and three out of the four cases (three data sets by us and one by Siy and Votta [113] that was further analyzed in article V) the share was around 10%. For the evolvability issue groups' structure and documentation, we could not find any consistencies. We also found that majority of the issues found in code reviews after initial functional testing were evolvability issues. It is difficult to assess the value of these empirical findings when there are only a limited number of studies to compare them with. However, based on the work to date, we can suggest that code reviews are a good tool for evolvability evaluation and improvement. Furthermore, it appears that evolvability issues are mostly about code structure and documentation rather than simple layout issues.

Third, we studied the human aspects in source code evolvability evaluations. We found that demographics such as education and general work experience are not good predictors of the evolvability evaluations. However, based on limited evidence, it seems that a person's role in the organization and the relationship (authorship) to the code have more effect on the evaluations than general demographics. This indicates that a person's organizational background should be taken into account when evaluating the reliability of the evaluations. Additionally, we were able to show that that the interrater agreement of the evolvability evaluations decreases for the more complex issues. Thus, there should be no need to double check whether a company's developer follows simple rules. However, for the more complex ones, there can be different interpretations. Therefore, double checking is a good idea when a new developer joins a company.

Fourth, we studied the relationship between human evolvability evaluations and source code metrics. When evaluating large software modules, human evaluations were somewhat conflicting with the respective source code metrics. However, when humans performed at method level for simple evolvability issues the code metrics human evaluations were highly correlated. Furthermore, we found that code metrics were less correlated to the human decision whether to refactor a code. This result is partly explained by our qualitative analysis of the evolvability issues detected by humans. There we found that humans are able to find issues from the source code that cannot be measured and detected with source code metrics. For example, if a developer’s code is well programmed from the engineering viewpoint, but it implements a functionality that is already available through a class library, then a metrics based analysis considers the code correct. However, an experienced developer will notice that this code is redundant. Thus, the source code metrics tools, widely studied in academia, are a good aid for detecting simple evolvability issues, but they offer a limited view of the source code evolvability as a whole.

## 6.2 Future work

In this study, we found preliminary evidence that a person’s role and code authorship affects the evolvability evaluations. Besides our study, we have also witnessed anecdotal evidence supporting this. For example, student groups in our laboratory’s software project course often regarded their own work products as high quality. However, when a new student group starts to develop the software, the new group thinks that the original code (developed by a student group from previous year’s course) is poor quality. Thus, it appears that there is a connection between a person’s relationship with the code and the person’s opinion of the code quality. In the future, it would be interesting to see high quality research studying this phenomenon.

We know that poor evolvability leads to increased effort in future development [9, 28, 40, 53, 77, 88, 107]. We know that software evolvability explains roughly 25% to 38% of the software evolution costs [9, 10, 107]. We know that there are companies and development methodologies that emphasize the importance of software evolvability by performing continuous refactoring [16, 39]. Still, it is difficult for a company to get reliable measures of the costs and benefits of software evolvability (e.g., for current software, will it pay off in the next two years if 10% of development is budgeted to software evolvability improvement). Therefore, to be able to demonstrate the possible costs and payoffs, we would need cost benefit models based on empirical data. Currently, we know that evolvability issues increase development effort, but we have no idea of the costs versus the benefits.  Future work should address this challenging topic.

This study also found solutions approach issues that had not been previously recognized in the software engineering literature. Future work should continue to study these issues, to confirm our findings that such issues really exist, to understand them better, to study them at the design level, to find better generalizations and possible fixes, and to see what their impact is on software development. By having a good understanding about these issues it should be possible to reduce problems in the software since the developers would be more aware of them and less likely to create such code and more likely to remove the problems.

In this study, we studied the distributions of code review defects. Similarly, we think that the quality impact of quality assurance methods, such as unit testing and acceptance testing, merit further study. A recent study of test-first unit testing indicates that it improves software design by creating code that has smaller units with less complexity [66]. Having broad knowledge of the quality impact, the defect types detected by different quality assurance methods would help software engineering practitioners choose the right tools for their quality assurance toolboxes depending on their quality goals.

Furthermore, we would like to see a stronger connection between people making code metrics tools and the developers working with actual code. In the past years, the study of code metrics has often started in the wrong direction. For example, researchers have come up with metrics and then started to think about the possible uses for it. In the future, it should be the other way around. Developers would identify the problems in the code and then researchers would create tools for detecting such code. Some work has already been done to implement these adaptive design flaw detectors [83, 84, 90, 109], and hopefully these tools will be integrated into mainstream development tools in future years.

## 7. REFERENCES

[1] AFOTEC, *Software Maintainability Evaluation Guide,* DEPARTMENT OF THE AIR FORCE, HQ Air Force Operational Test and Evaluation Center, 1996.

[2] Anda,B.C.D., "Assessing Software System Maintainability using Structural Measures and Expert Assessments," *in the 23rd International Conference on Software Maintenance (ICSM 2007),* 2007, pp. 204-213.

[3] Arisholm,E. and Sjoberg,D.I.K., "Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software," *Software Engineering, IEEE Transactions on,* vol. 30, no. 8, 2004, pp. 521-534.

[4] Arisholm,E., Sjøberg,D.I.K. and Jørgensen,M., "Assessing the Changeability of two Object-Oriented Design Alternatives--a Controlled Experiment," *Empirical Software Engineering,* vol. 6, no. 3, 2001, pp. 231-277.

[5] Arisholm,E., "Empirical Assessment of the Impact of Structural Properties on the Changeability of Object-Oriented Software," *Information and Software Technology,* vol. 48, no. 11, 2006, pp. 1046-1055.

[6] Arisholm,E., Briand,L.C., Hove,S.E. and Labiche,Y., "The Impact of UML Documentation on Software Maintenance: An Experimental Evaluation," *Software Engineering, IEEE Transactions on,* vol. 32, no. 6, 2006, pp. 365-381.

[7] Arnold,R.S., "Software restructuring," *Proceedings of the IEE,* vol. 77, no. 4, 1989, pp. 607-617.

[8] Balazinska,M., Merlo,E., Dagenais,M., Lague,B. and Kontogiannis,K., "Advanced clone-analysis to support object-oriented system refactoring," *in Proceedings of Seventh Working Conference on Reverse Engineering,* 2000, pp. 98-107.

[9] Bandi,R.K., Vaishnavi,V.K. and Turk,D.E., "Predicting maintenance performance using object-oriented design complexity metrics," *IEEE Trans. Software Eng.,* vol. 29, no. 1, 2003, pp. 77-87.

[10] Banker,R.D., Datar,S.M., Kemerer,C.F. and Zweig,D., "Software complexity and maintenance costs," *Commun ACM,* vol. 36, no. 11, 1993, pp. 81-94.

[11] Basili,V.R. and Selby,R.W., "Comparing the Effectiveness of Software Testing Strategies," *IEEE Trans.Software Eng.,* vol. 13, no. 12, 1987, pp. 1278-1296.

[12] Basili,V.R., Briand,L.C. and Melo,W.L., "A Validation of Object Oriented Design Metric as Quality Indicators," *IEEE Trans. Software Eng.,* vol. 22, no. 10, 1996, pp. 751-761.

[13] Bass,L., Clements,P. and Kazman,R., *Software Architecture in Practice,* Boston: Addison-Wesley, 2003.

[14] Beck,K. and Cunningham,W., "A laboratory for teaching object oriented thinking," *in Proceedings of Object Oriented Programming Systems Languages and Applications,* 1989, pp. 1-6.

[15] Beck,K., *Test-Driven Development by Example,* Addison-Wesley, 2002.

[16] Beck,K., *Extreme Programming Explained,* Canada: Addison-Wesley, 2000.

[17] Beizer,B., *Software testing techniques,* Van Nostrand Reinhold Co. New York, NY, USA, 1990.

[18] Binkley,A.B. and Schach,S.R., "Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures," 1998, pp. 452-455.

[19] Briand,L.C., Daly,J.W. and Wüst,J., "A Unified Framework for Cohesion Measurement in Object-Oriented Systems," *Empirical Software Engineering,* vol. 3, no. 1, 1998, pp. 65-117.

[20] Briand,L.C., Bunse,C. and Daly,J.W., "A Controlled Experiment for Evaluating Quality Guidelines on the Maintainability of Object Oriented Designs," *IEEE Trans. Software Eng.,* vol. 27, no. 6, 2001, pp. 513-530.

[21] Briand,L.C., Daly,J.W. and Wüst,J.K., "A Unified Framework for Coupling Measurement in Object-Oriented Systems," *IEEE Trans. Software Eng.,* vol. 25, no. 1, 1999, pp. 91-121.

[22] Briand,L.C. and Wüst,J.K., "Modeling development effort in object-oriented systems using designproperties," *Software Engineering, IEEE Transactions on,* vol. 27, no. 11, 2001, pp. 963-986.

[23] Briand,L.C., Wüst,J., Ikonomovski,S.V. and Lounis,H., "Investigating Quality Factors in Object-oriented Designs: an Industrial Case Study," in Proceedings of the 1999 International Conference on Software Engineering, 1999, pp. 345-354.

[24] Brooks,F.,P.Jr., *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition,* Addison Wesley Longman, Inc., 1999.

[25] Brown,W.J., Malveau,R.,C., McCormick,H.W. and Mowbray,T.,J., *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis,* New York: Wiley, 1998.

[26] Card,D.,N. and Glass,R.,L., *Measuring Software Design Quality*, Eaglewood Cliffs, New Jersey, USA: Prentice-Hall, 1990.

[27] Chaar,J.K., Halliday,M.J., Bhandari,I.S. and Chillarege,R., "In-process evaluation for software inspection and test," *Software Engineering, IEEE Transactions on,* vol. 19, no. 11, 1993, pp. 1055-1070.

[28] Chan,T.Z., Chung,S.L. and Ho,T.H., "An economic model to estimate software rewriting and replacement times," *IEEE Trans. Software Eng.,* vol. 22, no. 8, 08//. 1996, pp. 580-598.

[29] Chidamber,S.R., Darcy,D.P. and Kemerer,C.F., "Managerial use of metrics for object-oriented software: an exploratory analysis," *IEEE Trans. Software Eng.,* vol. 24, no. 8, 1998, pp. 629-639.

[30] Chidamber,S.R. and Kemerer,C.F., "A Metric Suite for Object Oriented Design," *IEEE Trans. Software Eng.,* vol. 20, no. 6, 1994, pp. 476-493.

[31] Chikofsky,E.,J. and Cross,J.,H., "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software,* vol. 7, no. 1, 1990, pp. 13-17.

[32] Chillarege,R., Bhandari,I.S., Chaar,J.K., Halliday,M.J., Moebus,D.S., Ray,B.K. and Wong,M.-., "Orthogonal defect classification-a concept for in-process measurements," *Software Engineering, IEEE Transactions on,* vol. 18, no. 11, 1992, pp. 943-956.

[33] Coad,P. and Yourdon,E., *Object-oriented analysis,* Upper Saddle River, NJ, USA: Prentice-Hall, 1991.

[34] Coad,P. and Yourdon,E., *Object-oriented design,* Englewood Cliffs, NJ: Prentice Hall, 1991.

[35] Cohen,J., "A coefficient of agreement for nominal scales," *Educational and Psychological Measurement,* vol. 20, no. 1, 1960, pp. 37-46.

[36] Coleman,D., Lowther,B. and Oman,P.W., "The Application of Software Maintainability Models in Industrial Software Systems," *J.Syst.Software,* vol. 29, no. 1, 1995, pp. 3-16.

[37] Coleman,D., Ash,D., Lowther,B. and Oman,P.W., "Using Metrics to Evaluate Software System Maintainability," *Computer,* vol. 27, no. 8, 1994, pp. 44-49.

[38] Cunningham,W., "The WyCash portfolio management system," *in Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum),* 1992, pp. 29-30.

[39] Cusumano,M.A. and Selby,R.W., *Microsoft Secrets,* USA: The Free Press, 1995.

[40] Darcy,D.P., Kemerer,C.F., Slaughter,S.A. and Tomayko,J.E., "The Structural Complexity of Software: An Experimental Test," *Software Engineering, IEEE Transactions on,* vol. 31, no. 11, 2005, pp. 982-995.

[41] Darcy,D.P. and Ma,M., "Exploring Individual Characteristics and Programming Performance: Implications for Programmer Selection," *in Proceedings of the 38th Annual Hawaii International Conference on System Sciences,* 2005, pp. 314a-314a.

[42] Deligiannis,I., Stamelos,I., Angelis,L., Roumeliotis,M. and Shepperd,M., "A controlled experiment investigation of an object-oriented design heuristic for maintainability," *The Journal of Systems & Software,* vol. 72, no. 2, 2004, pp. 129-143.

[43] Demeyer,S., Ducasse,S. and Nierstrasz,O., "Finding refactorings via change metrics," in Proceedings of the conference on Object-oriented programming, systems, languages, and applications, 2000, pp. 166-177.

[44] Ducasse,S., Rieger,M. and Demeyer,S., "A language independent approach for detecting duplicated code," *in Proceedings of the International Conference on Software Maintenance,* 1999, pp. 109-118.

[45] El Emam,K. and Wieczorek,I., "The repeatability of code defect classifications," *in International Symposium on Software Reliability Engineering,* 1998, pp. 322-333.

[46] El Emam,K., Melo,W. and Machado,J.,C., "The Prediction of Faulty Classes Using Object-Oriented Design Metrics," *The Journal of Systems and Software,* vol. 56, no. 1, 2001, pp. 63-75.

[47] Fenton,N.,E. and Ohlsson,N., "Quantitave Analysis of Faults and Failures in a Complex Software System," *IEEE Trans. Software Eng.,* vol. 26, no. 8, 2000, pp. 797-814.

[48] Fenton,N.,E. and Pfleeger,S.,L., *Software Metrics,* USA: Thomson Publishing Inc., 1996.

[49] Fowler,M., *Refactoring: Improving the Design of Existing Code,* Boston: Addison-Wesley, 2000.

[50] Fowler,M. and Beck,K., "Bad Smells in Code," in *Refactoring: Improving the Design of Existing Code,* 1st ed., Boston: Addison-Wesley, 2000, pp. 75-88.

[51] Genero,M., Piatini,M. and Manso,E., "Finding "early" indicators of UML class diagrams understandability and modifiability," *in Proceedings of International Symposium on Empirical Software Engineering,* 2004, pp. 207-216.

[52] Genero,M., Piattini,M. and Calero,C., "Empirical validation of class diagram metrics," *in Proceedings of the International Symposium on Empirical Software Engineering,* 2002, pp. 195-203.

[53] Gorla,N., Benander,A.C. and Benander,B.A., "Debugging effort estimation using software metrics," *Software Engineering, IEEE Transactions on,* vol. 16, no. 2, 1990, pp. 223-231.

[54] Grady,R.B., *Practical software metrics for project management and process improvement,* Prentice Hall Englewood Cliffs, NJ, 1992.

[55] Grady,R.B. and Caswell,D.L., *Software Metrics: Establishing a Company-wide Program,* Englewood Cliffs, NJ: Prentice Hall, 1987.

[56] Halstead,M.,H., *Elements of software science,* New York: Elsevier, 1977.

[57] Hamer,P.,G. and Frewin,G.,D., "M.H. Halstead's Software Science - a critical examination," in Proceedings of the 6th international conference on Software engineering, 1982, pp. 197-206.

[58] Harrison,R., Counsell,S.J. and Nithi,R.V., "An evaluation of the MOOD set of object-oriented software metrics," *IEEE Trans. Software Eng.,* vol. 24, no. 6, 1998, pp. 491-496.

[59] Henderson-Sellers,B., *Object-Oriented Metrics,* Upper Saddle River, New Jersey, USA: Prentice Hall, 1996.

[60] Hitz,M. and Montazeri,B., "Chidamber and Kemerer's metrics suite: a measurement theory perspective," *IEEE Trans. Software Eng.,* vol. 22, no. 4, 1996, pp. 267-271.

[61] Humphrey,W.S., *A Discipline for Software Engineering,* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

[62] Hunt,A. and Thomas,D., *The Pragmatic Programmer: From Journeyman to Master,* Boston: Addison-Wesley Professional, 1999.

[63] IEEE, "IEEE standard classification for software anomalies." *IEEE Std 1044-1993,* 1994.

[64] IEEE, *IEEE Standard Glossary of Software Engineering Terminology,* New York: The Institute of Electrical and Electronics Engineers, Inc., 1990.

[65] Iio,K., Furuyama,T. and Arai,Y., "Experimental analysis of the cognitive processes of program maintainers during software maintenance," *in Proceedings of International Conference on Software Maintenance.* 1997, pp. 242-249.

[66] Janzen,D., "Does Test-Driven Development Really Improve Software Design Quality?" *Software, IEEE,* vol. 25, pp. 77-84, 2008.

[67] Kafura,D.G. and Reddy,G.R., "The Use of Software Complexity Metrics in Software Maintenance," *IEEE Trans. Software Eng.,* vol. 13, no. 3, 1987, pp. 335-343.

[68] Kaner,C., Falk,J. and Nguyen,H.Q., *Testing Computer Software,* New York, NY, USA: John Wiley & Sons, 1999.

[69] Kataoka,Y., Ernst,M.D., Griswold,W.G. and Notkin,D., "Automated support for program refactoring using invariants," *in Proceedings of International Conference on Software Maintenance,* 2001, pp. 736-743.

[70] Kataoka,Y., Imai,T., Andou,H. and Fukaya,T., "A Quantative Evaluation of Maintainability Enhancement by Refactoring," *in Proceedings of the International Conference on Software Maintenance,* 2002, pp. 576-585.

[71] Kendall,M., Sir, "The problem of *m* ranking," in *Rank Correlation Methods,* 5th ed., J.D. Gibbons Ed. London: Edward Arnold, 1948, pp. 117-143.

[72] Kernighan,B.W. and Plauger,P.J., *The Elements of Programming Style,* New York, NY, USA: McGraw-Hill, Inc., 1978.

[73] LaToza,T.D., Venolia,G. and DeLine,R., "Maintaining mental models: a study of developer work habits," *in ICSE '06: Proceeding of the 28th international conference on Software engineering,* 2006, pp. 492-501.

[74] Lehman,M.M., "Laws of Software Evolution Revisited," *in Proceedings of European Workshop on Software Process Technology,* 1996, pp. 108-124.

[75] Lehman,M.M., "On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle," *The Journal of Systems and Software,* vol. 1, 1980, pp. 213-221.

[76] Li,W. and Henry,S.M., "Maintenance metrics for the object oriented paradigm," in Proceedings of the First International Software Metrics Symposium, 1993, pp. 52-60.

[77] Li,W. and Henry,S.M., "Object-Oriented Metrics that Predict Maintainability," *J.Syst.Software,* vol. 23, no. 2, 1993, pp. 111-122.

[78] Lieberherr,K.J. and Holland,I.M., "Assuring good style for object-oriented programs," *Software, IEEE,* vol. 6, no. 5, 1989, pp. 38-48.

[79] Lind,R.K. and Vairavan,K., "An experimental investigation of software metrics and their relationship to software development effort," *Software Engineering, IEEE Transactions on,* vol. 15, no. 5, 1989, pp. 649-653.

[80] Linnaeus,C. and Gmelin,J.F., *Systema naturae per regna tria naturae, secundum classes, ordines, genera, species, cum characteribus, differentiis, synonymis, locis,* Laurentius Salvius, 1758.

[81] Lorenz,M. and Kidd,J., *Object-Oriented Software Metrics,* Upper Saddle River, New Jersey, USA: Prentice Hall, 1994.

[82] Mäntylä,M.V., Vanhanen,J. and Lassenius,C., "Bad smells - Humans as code critics," *in Proceedings.20th IEEE International Conference on Software Maintenance, 2004.* 2004, pp. 399-408.

[83] Marinescu,R., "Measurement and Quality in Object-Oriented Design," *in Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005.ICSM'05.* 2005, pp. 701-704.

[84] Marinescu,R., "Detection Strategies: Metrics-Based Rules for Detecting Design Flaws," *in In Proceedings of Software Maintenance,* 2004, pp. 350-359.

[85] McCabe,T.J., "A Complexity Measure," *IEEE Trans. Software Eng.,* vol. 2, no. 4, 1976, pp. 308-320.

[86] McConnell,S., "High-Quality Routines " in *Code Complete 2,* 2nd ed., Redmond, Washington, USA: Microsoft Press, 2004, pp. 161-186.

[87] Mens,T. and Tourwe,T., "A survey of software refactoring," *IEEE Trans. Software Eng.,* vol. 30, no. 2, 2004, pp. 126-139.

[88] Miara,R.J., Musselman,J.A., Navarro,J.A. and Shneiderman,B., "Program indentation and comprehensibility," *Commun ACM,* vol. 26, no. 11, 1983, pp. 861-867.

[89] Miles,M.B. and Huberman,M.A., *Qualitative Data Analysis,* Thousand Oaks, California, USA: Sage Publications, 1994.

[90] Moha,N., Gueheneuc,Y.G. and Leduc,P., "Automatic Generation of Detection Algorithms for Design Defects," *in Automated Software Engineering, 2006. ASE '06. 21st IEEE/ACM International Conference on*
, 2006, pp. 297-300.

[91] Moilanen,T. and Roponen,S., *Kvalitativiisen aineiston analyysi Atlas.ti-ohjelman avulla ("Analyzing qualitative data with Atlas.ti software),* Helsinki, Finland: Kuluttajatutkimuskeskus, 1994.

[92] Munson,J.C. and Khoshgoftaar,T.M., "The detection of fault-prone programs," *Software Engineering, IEEE Transactions on,* vol. 18, no. 5, 1992, pp. 423-433.

[93] Murphy,G.C., Kersten,M. and Findlater,L., "How Are Java Software Developers Using the Eclipse IDE?" *Software, IEEE,* vol. 23, no. 4, 2006, pp. 76-83.

[94] Muthanna,S., Stacey,B., Kontogiannis,K. and Ponnambalam,K., "A maintainability model for industrial software systems using design level metrics," *in Proceedings of Seventh Working Conference on Reverse Engineering,* 2000, pp. 248-256.

[95] Nesi,P. and Querci,T., "Effort estimation and prediction of object-oriented systems," *The Journal of Systems & Software,* vol. 42, no. 1, 1998, pp. 89-102.

[96] Niiniluoto,I., "Käsitetyypit ja mittaaminen ("Concept types and measurement")," in *Johdatus tieteenfilosofiaan: Käsitteen ja teorianmuodostus ("Introduction to  philosophy of science: theory building and conception"),* 3rd ed., Helsinki: Otava, 1980, pp. 171-191.

[97] Oman,P.W., Hagemeister,J. and Ash,D., "A Definition and Taxonomy for Software Maintainability," Software Engineering Test Lab, University of Idaho., Tech. Rep. 91-08, 1991.

[98] Oman,P.W. and Cook,C.R., "The book paradigm for improved maintenance," *IEEE Software,* vol. 7, no. 1, 1990, pp. 39-45.

[99] Oman,P.W. and Cook,C.R., "Typographic style is more than cosmetic," *Commun ACM,* vol. 33, no. 5, 1990, pp. 506-520.

[100] Oman,P.W. and Hagemeister,J., "Constructing and testing of polynomials predicting software maintainability," *Journal of Systems and Software,* vol. 24, no. 3, 1994, pp. 251-266.

[101] O'Neill,D. , "National Software Quality Experiment Resources and Results," 2002, Accessed 2007 06/13 http://members.aol.com/ONeillDon/nsqe-results.html

[102] Parnas,D.L., "On the Criteria to Be Used in Decomposing Systems into Modules," *Communication of ACM,* vol. 15, no. 12, December/1972. 1972, pp. 1053-1058.

[103] Pigoski,T.M., *Practical Software Maintenance,* John Wiley & Sons Inc., 1996.

[104] Pocius,K.E., "Personality factors in human-computer interaction: A review of the literature," *Computers in Human Behavior,* vol. 7, no. 3, 1991, pp. 103-135.

[105] Porter,A.A., Votta,L.G.,Jr. and Basili,V.R., "Comparing detection methods for software requirements inspections: a replicated experiment," *Software Engineering, IEEE Transactions on,* vol. 21, 1995, pp. 563-575.

[106] Rising,L.S. and Calliss,F.W., "An information-hiding metric," *The Journal of Systems and Software,* vol. 26, no. 3, 1994, pp. 211-220.

[107] Rombach,D.,H., "Controlled Experiment on the Impact of Software Structure on Maintainability," *IEEE Trans. Software Eng.,* vol. 13, no. 3, 1987, pp. 344-354.

[108] Runeson,P. and Wohlin,C., "An Experimental Evaluation of an Experience-Based Capture-Recapture Method in Software Code Inspections," *Empirical Software Engineering,* vol. 3, no. 4, 1998, pp. 381-406.

[109] Schwanke,R.W. and Hanson,S.J., "Using Neural Networks to Modularize Software," *Mach.Learning,* vol. 15, no. 2, 1994, pp. 137-168.

[110] Seaman,C.B., "Qualitative methods in empirical studies of software engineering," *Software Engineering, IEEE Transactions on,* vol. 25, no. 4, 1999, pp. 557-572.

[111] Shepperd,M.J., "System architecture metrics for controlling software maintainability," *in IEE Colloquium on Software Metrics,* 1990, pp. 4/1-4/3.

[112] Shneiderman,B., *Software Psychology: Human factors in Computer and Information Systems,* Cambridge, Massachusetts, USA: Winthrop Publishers, 1980.

[113] Siy,H. and Votta,L., "Does the modern code inspection have value?" *in International Conference on Software Maintenance,* 2001, pp. 281-289.

[114] So,S.S., Cha,S.D., Shimeall,T.J. and Kwon,Y.R., "An empirical evaluation of six methods to detect faults in software," *Software Testing, Verification & Reliability,* vol. 12, no. 3, 2002, pp. 155-171.

[115] Stevens,W.P., Myers,G.J. and Constantine,L.L., "Structured Design," *IBM Syst J,* vol. 13, no. 2, 1974, pp. 115-139.

[116] Subramaniam,V. and Hunt,A., *Practices Of An Agile Developer,* Raleigh, North Carolina, USA: Pragmatic Bookshelf, 2005.

[117] Succi,G., Pedrycz,W., Djokic,S., Zuliani,P. and Russo,B., "An Empirical Exploration of the Distributions of the Chidamber and Kemerer Object-Oriented Metrics Suite," *Empirical Software Engineering,* vol. 10, no. 1, 01. 2005, pp. 81-104.

[118] Sun Microsystems. , "Code Conventions for the Java Programming Language," 1999, Accessed 1999 7/20 http://java.sun.com/docs/codeconv/

[119] Takahashi,D. , 2006, Accessed 2008 4/9 http://seattletimes.nwsource.com/html/businesstechnology/2003460386_btview04.html

[120] Tenny,T., "Program readability: procedures versus comments," *Software Engineering, IEEE Transactions on,* vol. 14, no. 9, 1988, pp. 1271-1279.

[121] Van Rysselberghe,F. and Demeyer,S., "Reconstruction of successful software evolution using clone detection," *Software Evolution, 2003.Proceedings.Sixth International Workshop on Principles of,* 2003, pp. 126-130.

[122] Wake,W.C., *Refactoring Workbook,* Addison Wesley, 2003.

[123] Welker,K.D., Oman,P.W. and Atkinson,G.G., "Development and application of an automated source code maintainability index," *Journal of Software Maintenance: Research and Practice*, vol. 9, no. 3, 1997, pp. 127-159.

[124] Weyuker,E.J., "Evaluating software complexity measures," *IEEE Trans. Software Eng.*, vol. 14, no. 9, 1988, pp. 1357-1365.

[125] Xing,Z. and Stroulia,E., "Refactoring Practice: How it is and How it Should be Supported-An Eclipse Case Study," *in Proceedings of the 22nd IEEE International Conference on Software Maintenance 2006*, 2006, pp. 458-468.

[126] Yu,H., Ikeda,M. and Mizoguchi,R., "Helping novice programmers bridge the conceptual gap," *in Proceedings of International Conference on Expert Systems for Development*, 1994, pp. 192-197.

[127] Yu,P., Systä,T. and Müller,H., "Predicting fault-proneness using OO metrics. An industrial case study," in Proceedings of Sixth European Conference on Software Maintenance and Reengineering, 2002, pp. 99-107.

ISBN 978-951-22-9856-3
ISBN 978-951-22-9857-0 (PDF)
ISSN 1795-2239
ISSN 1795-4584 (PDF)