Empirical Software Evolvability – Code Smells and Human Evaluations

Mika V. Mäntylä
SoberIT, Department of Computer Science
School of Science and Technology, Aalto University
P.O. Box 19210, FI-00760 Aalto, Finland
mika.mantyla@tkk.fi

Abstract—**Low software evolvability may increase costs of software development for over 30%. In practice, human evaluations and discoveries of software evolvability dictate the actions taken to improve the software evolvability, but the human side has often been ignored in prior research. This dissertation synopsis proposes a new group of code smells called the solution approach, which is based on a study of 563 evolvability issues found in industrial and student code reviews. Solution approach issues require re-thinking of the existing implementation rather than just reorganizing the code through refactoring. This work also contributes to the body of knowledge about software quality assurance practices by confirming that 75% of defects found in code reviews affect software evolvability rather than functionality. We also found evidence indicating that context-specific demographics, i.e., role in organization and code ownership, affect evolvability evaluations, but general demographics, i.e., work experience and education, do not**

Keywords-Doctoral dissertation synopsis; code smells; empirical study; code review; human evaluation; software maintainability;

I. Introduction

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. It has been long recognized that almost all large and successful software systems and products need continuous evolution. Brooks [1] stated that "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."

This study is about *software evolvability*, a quality attribute that reflects how easy software is to understand, modify, adapt, correct, and develop further. Empirical studies [2-4] have found that the added effort due to lack of evolvability varies between 25-36%. Although software evolvability has been studied extensively, the human evaluation of software evolvability has received considerably less attention. In addition, the types of evolvability issues found in-vivo have been mostly ignored while the focus is on evolvability criteria proposed by experts, e.g., design principles [5] and code smells [6].

This doctoral dissertation synopsis presents empirical research on code-level evolvability issues, i.e., code smells, and human evaluations of them. This work involves two research areas. First, it looks at types of software evolvability issues found in industrial and student settings. Furthermore, a classification was created based on the empirically discovered evolvability issues and the code smells presented in the literature. Second, this is a study of human evaluations of software evolvability using student experiments and industrial surveys. This paper is organized as follows. Section 2 positions the work and outlines the main concepts in the research space. Section 3 presents the research questions and methods. Next, answers to research questions are provided in Section 4. Finally, Section 5 provides the conclusions and outlines directions for further work.

II. Dissertation Research Space

Figure 1 illustrates the topics covered in the literature review of the thesis overview [7] and shows how our research questions link to the relevant topics (research questions are presented in Section 3). Software evolvability can be operationalized with software evolvability criteria, which have been largely created based on 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, which are also a subset of software evolvability criteria. Thus, the dissertation first 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 increase the benefits of applying these criteria. The only study that the author is aware of that focused on evolvability issues detected in-vivo by humans was [8] that studied the types of evolvability issues identified in code reviews. Even that study did not contain a detailed analysis of the evolvability issues found.

The second 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 evolvability issue to be a problem, then that individual is not likely to remove this problematic issue from the software. Therefore, differences in human evaluations can lead to differences 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 prior studies, human evaluations are not the primary focus, 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. Additionally, some of the recently suggested evolvability issues, i.e., the code smells [6], had not been studied empirically when the research was started.

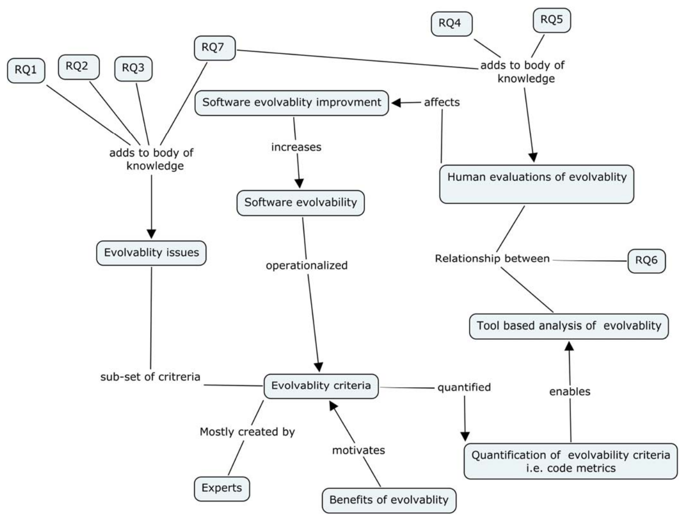

Figure 1 Topics covered in existing literature and research questions.

III. Methodology

In this section the research questions are presented and the research methods are summarized.

A. Research questions

The dissertation consists of seven research questions, five publications referred to as papers 1-5 [9] [10] [11] [12] [13], and the dissertation overview [7]. Next, we present the research questions and the publications in which they are addressed.

- **RQ1** paper 5 [13]: What is the distribution of evolvability issues and functional defects found in code reviews?
- **RQ2** papers 1, 2, 4, 5 [9, 10, 12, 13]: How can the evolvability issues, either presented in the literature or identified by humans, be classified?
- **RQ3** papers 4 and 5 [12, 13]: What types of evolvability issues are identified in the source code by humans and how are they distributed to different types?
- **RQ4** papers 2 and 3 [10, 11]: Do humans achieve interrater agreement when performing code evolvability evaluations?
- **RQ5** papers 2 and 3 [10, 11]: Do the demographics of humans affect or explain the evolvability evaluations, and if so, how?
- **RQ6** papers 2-4 [10-12]: What is the relationship between evolvability evaluations and source code metrics, do the evaluations and metrics correlate or explain each other?
- **RQ7** paper 4 [12]: What evolvability issues are seen as the most significant by human evaluators?

B. Research Methods

Table 1 summarizes the research methods used in 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 4 was an uncontrolled experiment, and the data was collected through a survey, so it is labeled S/E. Furthermore, N&L means that both qualitative and quantitative data analysis were major factors in the results. One of the strengths of the dissertation is that the use of both students and industrial workers provides a nice variety of subjects and reduces the possibility of bias.

Table 1 Summary of research methods

	Article				
	Paper 1 [9]	*Paper 2* [10]	*Paper 3* [11]	*Paper 4* [12]	*Paper 5* [13]
Data source	I	I	S	S	I&S
Data collection	S	S	E	S/E	O
Data analysis	N	N	N	N&L	L

IV. Answers for Research Questions

This section provides a brief answer for each research question. For more detailed answers, the reader should access and study the original publications.

A. RQ1: Code review defect types

What is the distribution of evolvability issues and functional defects found in code reviews? Based on our paper 5 [13], 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 [8] had previously proposed: based on data from a single company, most code review findings are evolvability issues.

We recognize that quality assurance made prior to the code review can have a significant impact on the results, e.g., 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 in the industry. Unfortunately, [8] did not reveal what quality assurance activities were performed prior to code review in their study.

To determine whether the large number of evolvability issues was simply due to chance, we re-analyzed public data available from prior code review studies. Unfortunately, we found only four studies [14-17] with a sufficient number of defects and enough information to try to perform an approximation of the defect distributions. Figure 2 compares eight data sets, showing the proportions of functional defects and evolvability issues. To summarize, in the figure we have three data sets (Mi, Ms, Si) used by the authors to compare the proportions of functional defects and evolvability issues, and these data sets show that the proportion of evolvability issues are between 76 and 86%. In addition, five other data sets (O, So, E1, E2, C) are more unreliable, as they have not specifically assessed this research question, produced mixed results.

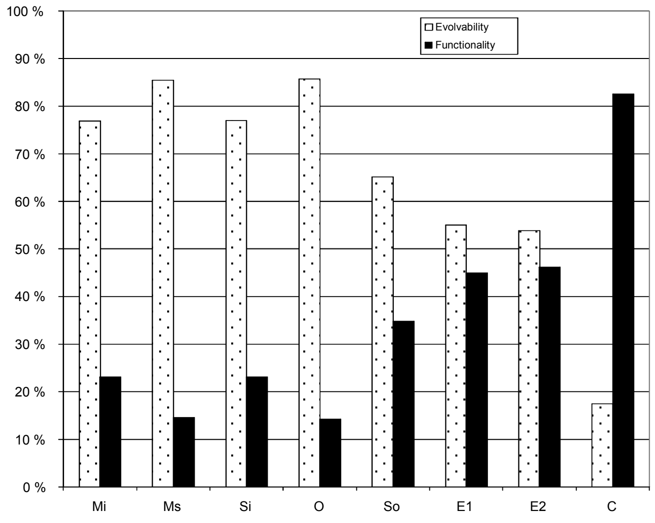

Figure 2. Defect distribution of functional and evolvability issues: Mi and Ms [13], Si [8], O [14], So [17], E1 and E2 [15], and C [16]

B. RQ2: Classification of evolvability issues

How can the evolvability issues, either presented in the literature or identified by humans, be classified? Figure 3 presents the classification of evolvability issues. The first three levels of the classification are based on the 563 evolvability issues classified in paper 5 [13]. The lowest level of the classification is based on the code smells by Fowler and Beck [6], and this smells classification was originally published in paper 1 [9].

In classifying evolvability issues, we have three main types: *documentation, visual representation*, and *structure*. 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.

The documentation was further divided into 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 concerning code element naming and code commenting.

The structure group was divided into subgroups: *re-organize* and *solution approach*. 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 method of implementation that often requires only a limited amount of structural modification. For example, knowing the existence of prebuilt functionality that can be used instead of using a self-programmed implementation and replacing a program's array data structure with a vector would be considered a solution approach issue. Therefore, solution approach recommendations are not related to reorganizing existing code, but they refer to rethinking the current solution and implementing it in a different way.

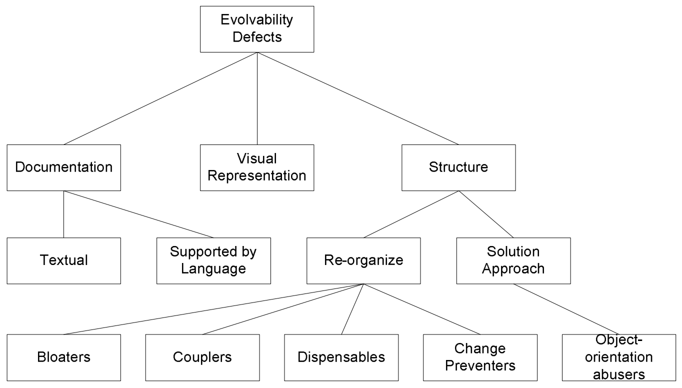

Figure 3 Classification of evolvability issues

The distinction between two of the structural evolvability issue types - solution approach and reorganize - is perhaps one of the most important findings. Reorganize issues have been widely recognized and studied in the past. Fixing the reorganize issues has been studied extensively by researchers who use the term refactoring [18]. Furthermore, the automatic detection of reorganization issues has been studied through static code analyzers and code metrics tools. Prior studies in the software engineering domain have not focused on or discussed 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.

At the lowest level, i.e., under textual, reorganize, and solution approach subgroups, we have classified the code smells [6] into six different groups as follows. The *bloater* smells represent something that has grown so large that it cannot be effectively handled (long method, large class, primitive obsession, long parameter list, and data clumps from [6]). The *object-orientation abusers* represent cases in which the solution does not fully exploit the possibilities of object-oriented design (switch statements, temporary field, refused bequest, and alternative classes with different interfaces). The *change preventers* are smells that hinder changing or further developing the software (divergent change, shotgun surgery, and parallel inheritance hierarchies). These smells violate the rule suggested by Fowler and Beck [6], which states that classes and possible changes should have a one-to-one relationship (pp. 80). The *dispensables* represent something unnecessary that should be removed from the source code (lazy class, data class, duplicate code, dead code, and speculative generality). The *couplers* are four coupling-related smells (feature envy, inappropriate intimacy, message chains, and middleman). One design principle that has been around for decades is low coupling [5]. 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.

C. RQ3: Distribution of evolvability issues

What types of evolvability issues are identified in the source code by humans and how are they distributed to different evolvability issues? In Table 2, the distribution of evolvability issues is presented. In the code reviews (paper 5 [13]), there were 276 and 287 evolvability issues, and in the refactoring experiment (paper 4 [12]), 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, about 15%, in the refactoring experiment. Roughly one third of the evolvability issues in 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.

It is difficult to find any consistencies in the shares of the evolvability issues in Table 2, outside of the share of visual representation issues. In our three data sets, visual representation issues had the most constant and lowest proportion of defects with percentages varying from 9.8% to 14.7%. In [8], visual representation had a share of 11%. Thus, based on the four sources of evidence, it seems that visual representation accounts for a consistent 10% share of evolvability 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, different code files were reviewed. Thus, a possible bias that came from having several structural issues in the refactoring experiment code may have caused the radical differences in the shares of structural issues when compared to the code reviews.

Table 2 Distribution of evolvability issues

Type	Industrial Reviews		Student Reviews		Refactoring experiment[a]	
Documentation	96	34.8%	132	46.0%	62	25.1%
Visual Rep.	27	9.8%	31	10.8%	35	14.7%
Structure	153	55.4%	124	43.2%	190	75.7%
Total	276	100.0%	287	100.0%	245	100%

The sum of issues may exceed 100%; see [7, 12] for details.

D. RQ4: Interrater agreement

Do humans achieve interrater agreement when performing code evolvability evaluations? We found high interrater agreement for simple code smells such as method length. We found lower interrater agreement for complex code smells, such as feature envy. It seems natural that agreement is higher on the simple issues than it is on the ones that are more complex. (papers 2 and 3 [10, 11])

Our results indicate that controlling and preventing simple evolvability issues can be achieved by coding rules and tool based checking of them. To control simple evolvability issues, it is not necessary to have senior developers check the code. 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 other quality control method 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.

E. RQ5: Demographics

Do the demographics of humans affect or explain the evolvability evaluations, and if so, how? In paper 2 [10], 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, and 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. Furthermore, the two original developers who had written much of the code considered the code to be much less "smelly" than the other developers.

Unfortunately, in paper 2 [10], only a small number of developers participated, so we could not utilize robust statistical methods. In paper 3 [11], 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. It seems that in paper 3 [11], the laboratory settings and the lack of personal attachment and ownership to the code that were present in paper 2 [10] affected the results. In paper 3 [11], we had no differences in evaluators' roles, knowledge, and code ownership and thus could not study them. To answer this research question, we concluded that people's relationships with the code and the organization (context-specific demographics) may affect the evaluation results, but people's general demographics, 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.

F. RQ6: Evaluations and code metrics

What is the relationship between evolvability evaluations and source code metrics; do the evaluations and metrics correlate or explain each other? In paper 2 [10], 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 paper 2 [10], the developers performed the evaluation on the modules on which they had primarily worked, but the answers were based on their recollections. In addition, in paper 2 [10], the number of developer evaluations was too small for statistical analysis.

To fix these shortcomings, paper 3 [11] used 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. In paper 4 [12], we continued our study on the relationship between evolvability issues and code metrics. In paper 4 [12], we studied the evolvability issues humans find in the source code and found that only some of these can be found with automatic tools.

The results from the three articles of the thesis 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. 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.

G. RQ7: Significant evolvability issues

What evolvability issues are seen as the most significant by human evaluators? This research question was studied in paper 4 [12] where we considered the evolvability issues found in the source code and linked them to the refactoring decision. We studied the detailed evolvability issue types and found that the long method code smells and the need to perform extract method refactoring, was the most important individual predictor of a refactoring decision. We also found that both positive and negative comments affect the refactoring decision. In practice, this means that if a method has poor evolvability it can balance it with having some positive aspects, e.g., good commenting to compensate long and complex structure.

V. Contributions and Further Work

Seventy-five percent of code review defects do not affect program execution but they do impact the evolvability of the software. Information about defect types detected with particular quality practices is important as it helps organizations to decide whether to use particular procedures. Code reviews seem to be the most useful method to examine new or emerging software products that have long lifetime ahead of them. The author would like to see similar studies about other quality practices. For example, [19] suggested that only 17% of faults found in document inspections would propagate into the code. Confirming or refuting such results would provide valuable information about document inspections.

Solutions approach issues represent a big challenge for tool-based evolvability measurement and highlight programming as a craft. An example of solution approach is knowing the existence of prebuilt functionality that should be used instead of using a self-programmed implementation. It is difficult to see how such issues could be detected with tools. I think the number of solutions approach issues will increase as software becomes even more layered and distributed and utilizes more code libraries. Thus, in the future the current evolvability measures will become even less useful unless someone finds a way to cope with such issues.

One's relationship with code and organization affects evolvability evaluations, but general demographics do not. In the company context, we found context-specific demographics that affected code evolvability evaluations, e.g., code ownership and role in the organization. In a lab setting, we studied the general demographics (e.g., education, work experience) and found that they had no effect on evolvability evaluations. This result seems intuitive, and it highlights the impact that context has on software development activities.

Humans and tools have high agreement on simple code smells but conflict with more complex ones. A practical implication of this result is that humans cannot be replaced with tool-based evolvability analysis. However, one should use tools to detect code smells that are simple, e.g., long method, as in such cases there is a high correlation between humans and tools. This leaves humans more time for other more productive work.

Acknowledgment

The author would like to thank the de-facto supervisor Casper Lassenius, official supervisor Reijo Sulonen, pre-examiners Erik Arisholm (Simula Research Laboratory), Tarja Systä (Tampere University of Technology), and opponent Magne Jørgensen (Simula Research Laboratory). Valuable help was given by all SPRG members and especially by Juha Itkonen, Kristian Rautiainen, Jari Vanhanen, and Jarno Vähäniitty.

References

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

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

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

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

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

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

[7] M.V. Mäntylä, Software Evolvability – Empirically Discovered Evolvability Issues and Human Evaluations, doctoral dissertation, Helsinki University of Technology, 2009, http://lib.tkk.fi/Diss/2009/isbn9789512298570/

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

[9] M.V. Mäntylä, J. Vanhanen and C. Lassenius, "A Taxonomy and an Initial Empirical Study of Bad Smells in Code," Proceedings of the International Conference on Software Maintenance, 2003, pp. 381-384.

[10] M.V. Mäntylä and C. Lassenius, "Subjective evaluation of software evolvability using code smells: An empirical study," Journal of Empirical Software Engineering, vol. 11, no. 3, 2006, pp. 395-431.

[11] M.V. Mäntylä, "An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement," International Symposium on Empirical Software Engineering, 2005, pp. 277-286.

[12] M.V. Mäntylä and C. Lassenius, "Drivers for Software Refactoring Decisions," International Symposium on Empirical Software Engineering, 2006, pp. 297-306.

[13] M.V. Mäntylä and C. Lassenius, "What Types of Defects Are Really Discovered in Code Reviews?" Software Engineering, IEEE Transactions on, vol. 35, no. 3, 2009, pp. 430-448.

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

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

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

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

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

[19] T. Berling and T. Thelin, "An industrial case study of the verification and validation activities," Software Metrics Symposium, 2003. Proceedings. Ninth International, 2003, pp. 226-238.