Mika V. Mäntylä and Casper Lassenius. 2009. What types of defects are really discovered in code reviews? IEEE Transactions on Software Engineering, accepted for publication.

What Types of Defects Are Really Discovered in Code Reviews?

Mika V. Mäntylä and Casper Lassenius, *Member*, *IEEE*

Abstract—Research on code reviews has often focused on defect counts instead of defect types, which offers an imperfect view of code review benefits. In this paper, we classified the defects of nine industrial (C/C++) and 23 student (Java) code reviews, detecting 388 and 371 defects, respectively. First, we discovered that 75 percent of defects found during the review do not affect the visible functionality of the software. Instead, these defects improved software evolvability by making it easier to understand and modify. Second, we created a defect classification consisting of functional and evolvability defects. The evolvability defect classification is based on the defect types found in this study, but, for the functional defects, we studied and compared existing functional defect classifications. The classification can be useful for assigning code review roles, creating checklists, assessing software evolvability, and building software engineering tools. We conclude that, in addition to functional defects, code reviews find many evolvability defects and, thus, offer additional benefits over execution-based quality assurance methods that cannot detect evolvability defects. We suggest that code reviews may be most valuable for software products with long life cycles as the value of discovering evolvability defects in them is greater than for short life cycle systems.

Index Terms—Code inspections and walkthroughs, enhancement, extensibility, maintainability, restructuring.

1 Introduction

Despite the fact that peer reviews have been extensively studied, knowledge of their benefits is still inadequate. Previous research on reviews has repeatedly shown them to be an effective quality assurance measure, catching a large percentage of overall software defects. For example, Boehm and Basili [12] noted that reviews catch 60 percent of product defects.

While understanding that the relative number of defects caught by different quality assurance methods is valuable, it provides only a limited view of their effectiveness unless the types of defects that various methods discover are understood. Thus, while formal experiments on reviews reporting only defect counts provide valuable data, it is difficult to assess the costs and benefits of reviews and inspections solely upon such data [40]. Similarly, industrial cases reporting only the number of defects per person hour have little value. In addition to not providing information about the types of defects discovered, defect counts can be misleading. There can be fundamental differences in the defect counts as the defect definition may vary from case to case. Thus, to better understand the effectiveness and value of various quality assurance methods, we must also comprehend the type of defects each method is likely to reveal. Only by understanding the types of defects identified, as well as the context and quality goals, can guidelines be proposed for the order and timing of various quality assurance methods.

• *The authors are with the Helsinki University of Technology, PO Box 9210 02015 TKK Finland. E-mail: {mika.mantyla, casper.lassenius}@tkk.fi.*

Manuscript received 4 Sept. 2007; revised 5 June 2008; accepted 4 Aug. 2008; published online 5 Aug. 2008.
Recommended for acceptance by D. Rombach.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2007-09-0257.
Digital Object Identifier no. 10.1109/TSE.2008.71.

While peer review literature has focused on defect counts and often overlooked qualitative issues, such as defect types and severities, the few existing studies that look deeper into types of defects provide some initial insight. For example, Siy and Votta [67] proposed that 75 percent of the defects found by code reviews are evolvability defects that affect future development efforts instead of runtime behavior. Despite this, most previous studies have focused on functional defects, i.e., defects that affect runtime behavior. Thus, the vast majority of defects found in code reviews seem to have been ignored in most published studies. While one may question the value of discovering evolvability defects, several studies from academia and industry [2], [4], [5], [13], [17], [23], [27], [30], [46], [51], [57], [62], [69] indicate that poor evolvability results in increased development effort in terms of lower productivity and greater rework.

Making a distinction between functional and evolvability defects is important for three reasons. First, code reviews are often compared to various testing techniques that cannot detect evolvability defects, giving an incomplete picture of the benefits of code reviews. Second, code review literature has created the impression that code reviews are mainly about finding functional defects. Third, the practical implications of the distribution between functional and evolvability defects are significant. For example, one would expect software product companies to be interested in discovering evolvability defects since their products may need to withstand heavy subsequent evolution, whereas software project companies would be less interested in evolvability issues. In addition, code reviews have other benefits, such as increased knowledge transfer and learning. Thus, looking at the types of defects found represents only a partial, but important, illustration of their benefits.

In this paper, we qualitatively explore and classify defects discovered in code reviews. Furthermore, we study the distributions of defect types to determine if there are generalizable consistencies. In many disciplines, but most notably in the natural sciences, classifications are created, maintained, and improved to increase scientific knowledge [56], such as Linnaeus's classification of the natural world [47] and the periodic table of the chemical elements. The classification of defects has five purposes. First, we hope that classification will increase the body of knowledge in software engineering and bring to light the true benefits of code reviews as well as their role in the software development process. Second, the classification can be useful when creating company coding standards and code review checklists, as well as assigning roles to participant of code reviews. Third, the classification of evolvability defects 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, memory handling defects are much less frequent in Java than in C language. Fifth, organizations may utilize defect classifications to receive feedback from the development process, allowing deeper learning from the mistakes than if using only defect counts. We hope to stimulate similar studies of other quality assurance methods, thus constructing a basis for making informed decisions on which quality assurance method or methods to choose for each real-world situation.

This paper is organized as follows: Section 2 introduces the relevant theoretical background. Section 3 delineates the methodology and Section 4 outlines the results. Finally, Sections 5 and 6 present the discussion and the conclusions.

2 Theoretical Background

This section summarizes previous research and identifies research space we try to fill. First, we discuss definitions of the term *defect* and define it as a deviation from quality. Second, we review existing defect taxonomies, discussing their common points and shortcomings. Third, we review existing empirical work on defects discovered in code reviews and find contradicting evidence. Some studies indicate that the majority of code review defects are evolvability defects, but still, many highly regarded studies have ignored these. Fourth, to understand whether evolvability issues are actually worth discovering and removing, we discuss their economic value.

2.1 Definition of a Defect

There seems to be no universally accepted definition of the term defect, even in the code review context. An often-quoted distinction is the *error*, *fault*, *failure* by IEEE [34]. However, in the code review literature, there are three different viewpoints of the term. The viewpoint is not dependent on the author, as the same author may utilize different viewpoints in different works. The viewpoints are as follows, in order of breadth:

- Defects as failures [42], [70]: This viewpoint emphasizes that only findings in the code should be counted as defects if they cause the system to fail or produce unsatisfactory results (i.e., produce failures) when executed.
- Defects as faults [14], [34], [64]: According to the IEEE [34], the term *fault* means an incorrect step, process, or data definition. Hence, this viewpoint considers defects as incorrect code, which may or may not result in a system failure.
- Defects as deviations from quality [29], [32], [44], [63]: This is clearly the broadest viewpoint, exemplified by the PSP guide [32, p. 38], which states, "A defect is counted every time you make a program change."

The above definitions are quite different and it seems possible that these differences can cause large differences in code review results, particularly with respect to defect counts. In this study, we explore the findings detected during the code review process. Thus, in order to consider all findings, to note bias the results, and to reveal the true nature of defects detected by code reviews, we have opted to use the broadest possible definition of defects (i.e., defects as deviations from quality). *Quality* is a term that is difficult to define precisely and such an attempt is largely dependent on one's point of view [28], [39]. We take a pragmatic point of view regarding quality and define it from the viewpoint of the code review team. Thus, if the code review team finds an issue and agrees that it is a deviation from quality, the issue is counted as a defect.

2.2 Defect Taxonomies

There is no shortage of defect taxonomies. According to Runeson et al. [64], defects are often classified according to three dimensions. First, a defect can be classified as either an *omission* or a *commission*. Second, a defect may be classified based on its *technical content*. Third, defects may be classified by the *impact* on the user.

In this work, we focus on defects' technical content and, thus, we identified and studied the major prior works in this area. The studied taxonomies were by Basili and Selby [7], Chillarege et al. [18], Humphrey [32, p. 262], Beizer [10], Kaner et al. [37], IEEE [33], and Grady [31]. A comparison of the taxonomies reveals that there are several similarities and that numerous classifications share many defect types. However, we also found all classifications to have shortcomings: The classifications either are missing defect types or use restrictive definitions. For example, Kaner et al. have perhaps the most extensive classification, but its missing defect type interface and its hardware defect type represent only part of the interface defects. Furthermore, evolvability defects have received little attention in the taxonomies. The existing defect classifications are discussed in depth in Section 4.3, where we construct our functional defect classification.

2.3 Code Review Defects

Software reviews have been extensively studied. However, very little information on the detected defect types was provided in the most recent review articles [3], [44], [64]. While suspecting that the same would be true for primary studies, we performed a literature search in the Inspec database using the terms (searching all fields) *code inspection* and *code review*, resulting in 96 and 66 records, respectively.

For these records, we first read the abstracts to determine whether the papers were likely to contain information about the defects discovered in code reviews. Furthermore, we searched for studies citing the most influential articles. In the literature study of code review defects, we excluded two types of studies. First, studies that did not provide or utilize a classification of discovered defects were excluded (for examples, see [3], section Inspection Metrics). Second, we excluded small-scale studies (those with less than 50 defects) as distributions from small data sets are likely to be biased. Next, we present the most influential prior works of code review defect types.

Fagan's work [26] provided a schema of defect classification and distribution, but the details of the classification are not presented, making it difficult to understand. Russell [65] presented the data of 2.5 million lines of inspected code, where defects were classified as major, minor, and commentary, making the classification quite coarse. Basili and Selby [7] provided well-defined and often cited 2D classification. Unfortunately, the number of defects in the study was only 34, limiting its usefulness as a source for defect distributions.

Chillarege et al. [18] created a defect classification consisting of eight defect types and a distribution of roughly 300 defects. Several studies have utilized their classification, but, unfortunately, there are large fluctuations in reported defect distributions. This is natural as studies are conducted in different environments. However, the fluctuations can also be due to different interpretations of the defect type scheme. For example, the defect type function is described quite differently in Chillarege et al. [18] than in Humphrey [32]. Thus, the differences between the operationalizations in the previous studies make quantitative comparison challenging.

In prior works of code review defects, evolvability defects have been mostly ignored. Chillarege et al. [18], Humphrey [32], and Runeson and Wohlin [63] mention only one evolvability defect type, *documentation*. However, we think that it might be possible, based on the defect type description, that Chillarege et al. [18] included evolvability defects as well as functionality defects in the function defect type of their classification. Additionally, El Emam and Wieczorek [25] have defect classes called *understandability* and *naming conventions*. However, none of these studies included defect types for structural evolvability defects, such as long methods, duplicate code, or incorrectly located functionality. Naturally, there are three possible causes: First, structural defects might have been ignored; second, they might have been classified under another defect class; or third, such defects did not exist in the code. As the last explanation seems very unlikely, the results from the study are most likely biased when it comes to the proportions of evolvability and functional defects.

Ignoring evolvability defects seems odd. Siy and Votta [67] suggested that, based on 130 code inspection sessions, the majority of code inspection findings are evolvability defects. In their study, only 18 percent of the findings identified were functional defects causing system failure, 22 percent were false positives, and 60 percent were evolvability defects. Their research further categorized 369 evolvability defects of 31 code inspection sessions and created four groups: *documentation*, *style*, *portability*, and *safety*. 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.

Siy and Votta's findings are also supported by a nonacademic report by O'Neill [58] indicating that 85 percent of the defects detected in code reviews do not affect execution. This report is based on over 3,000 code inspections that covered over one million lines of code. Although O'Neill's report is certainly interesting, it should be studied with caution, as the methodology and data collection mechanisms are only vaguely described. Furthermore, similar results have been reported for document reviews: An industrial study [11] found that only 17 percent of faults found in document inspections would propagate into the code.

We recognize the studies by Chillarege et al. [18] (elaborated in [16]) and Siy and Votta [67] as the most significant prior works on the qualitative analysis and classification of defects found in code reviews. Chillarege et al. created a classification that was used to classify a large amount of defects, which was later extended and utilized in studies by other scholars. Siy and Votta were the first to suggest that the majority of code review findings are evolvability defects.

2.4 Economic Value of Evolvability Defects

Prior work hints that the majority of code review defects might be evolvability defects. Practitioner literature on inspections (e.g., [29], [70]) has emphasized the identification of major issues that affect cost and quality. Naturally, one may argue that evolvability defects are not important as they will never be seen by the customer and they cannot directly cause a system failure. Thus, before studying evolvability defects in more detail, their value or impact must be assessed.

Several studies [4], [5], [13], [17], [23], [46], [62] showed poor structure results in increased development effort. Experiments by Bandi et al. [4] and Rombach [62] both compared two functionally equal systems that had different levels of structural evolvability. Bandi et al. showed that adding new functionality took 28 percent longer and fixing errors took 36 percent longer for the less evolvable system. Rombach's data indicated that requirement changes took 36 percent longer and fixing errors took 28 percent longer in the less evolvable system. Studies utilizing industrial data naturally do not have the luxury of comparing two functionally equal systems. Thus, they have used regression models [5], [17], [46] to show that poor software structure is

correlated with lower productivity and greater rework. Based on such models, Banker et al. [5] reported that software structure may account for up to 25 percent of the total maintenance costs. In addition to structure, visual representation and documentation have been shown to have an effect on program evolvability [30], [51], [57], [69]. Miara et al. [51] showed that two and four space indentations were correlated with the highest test scores when compared to zero and six space indentations. Oman and Cook [57] indicated that an advanced form of code commenting and layout (called *book paradigm* in the original work) is correlated with significantly higher comprehension test scores and slightly shorter test times when compared to traditional commenting and layout. The results of Tenny [69] indicated that code commenting has a larger effect on program comprehension than code structure. Evidence of the benefits of code element naming indicated that proper identifier name length is correlated with shorter debugging times [30]. Furthermore, studies of source code [54], [72] modifications showed 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.

In the industry, practitioners have realized that poor software evolvability has economic importance. For example, Microsoft's Office division has determined that 20 percent of the development effort should be budgeted to code modification [22]. An empirical study [45], conducted 11 years after the original study, showed that the actual time spent on making the code more maintainable was 15 percent of the development effort. Although this number is slightly smaller than what was claimed earlier, it shows that the idea is implemented in practice. Similarly, *extreme programming* [9], a software development method created by industry experts, has recognized the need for high software evolvability. Extreme programming emphasizes short development cycles with frequent releases, close collaboration with the customer, 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 to assure high evolvability. Industry experts [2], [27] who advocate refactoring claim that high evolvability results in easier program comprehension and defect detection, increased software development speed, better testing, auditing, and documenting, reduced dependency on individuals, greater job satisfaction, and extension of a system's lifetime, all of which preserve the software value for the organization. Finally, some industry experts view poorly evolvable code as technical debt that can slow down development. Thus, debt should be promptly paid to avoid high interest accruing when working with poorly evolvable code [21]. Although industrial sources mostly lack supporting data, it seems that industrial experts consider software evolvability important.

Finally, the potential economic value of evolvability defects is largely based on the context. If the software is facing several years of subsequent evolution, the costs of unfixed evolvability defects are likely to be high during the lifetime of the software. On the other hand, if the software is not modified or further developed after the code review, it makes very little sense to try to detect evolvability issues.

3 Methodology

In this section, we present the methodology of the study. First, we present the research questions and then we describe the research setting and the data analysis.

3.1 Research Questions

This study addresses the types and distributions of the defects found in code reviews. We divide this overall problem into two research questions. Our first question is of a confirmatory nature.

Research question 1. *What is the distribution between functional defects and evolvability defects?*

We wish to study the proposition by Siy and Votta [67] that most defects discovered in code reviews are evolvability defects rather than functional ones. We define a *functional defect* as *a defect in the code that may cause system failure when the code is executed*. This definition is identical to the fault definition given by the IEEE [34]. Further, we define an *evolvability defect* as *a defect in the code that makes the code less compliant with standards, more error-prone, or more difficult to modify, extend, or understand*. Virtually all evolvability defects can be claimed to make the code more error-prone. If error-prone codes were included in the definition of functional defect, there would not be any items in the evolvability defects' class.

Research question 2. *What different types of evolvability and functional defects are found in code reviews and how can they be classified based on the defects' technical content?*

This research question is both exploratory and confirmatory. For functional defects, there are several existing classifications. We first compared existing functional classifications and created a classification based on their defect classes. Second, we studied whether such defects are discovered in the reviews. For functional defects, our study is confirmatory, and for evolvability defects, it is exploratory. With technical content, we refer to a defect classification dimension by Runeson et al. [64] that emphasizes the technical nature of defects. Finally, we compared our defect classification and distributions with previously presented defect classifications and distributions.

3.2 Research Setting

The types of defects identified in code reviews might vary due to contextual variables, such as the code reviewed, the review process, the reviewers' focus, and the reviewers themselves. In this paper, we present data from two different environments, industrial and student code reviews, as a measure to improve validity. Naturally, having only two data sources, the sample is still limited, but, nevertheless, it is an improvement over many past studies that are based only on single data sources.

Quality assurance performed 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 [32], [63]. In our studies, developers performed automatic unit testing or quick

TABLE 1
Student Demographics

	Mean	Std. Dev	Median	Range
Credits (180 required for master's degree)	115	38.0	112	3-230
Work experience in years	1.6	2.47	0.5	0-15
Java understanding (1-5)	3.5[1]	0.86	3	1-5
Code review experience (1-5)	1.6	0.89	1	1-5

[1] *The authors are aware of the controversy relating to the use of mean values from ordinal scale data [35], [41].*

functional testing prior to code review, something we consider a typical scenario in the real world.

3.2.1 Industrial Code Reviews

We studied code reviews in one industrial company. The company was selected based on accessibility and the fact that it had an established code review practice. The company develops software products for professional engineering design, has several thousand customers, and roughly 100 people working in its software development department. We observed the code review sessions of six development teams, working with three separate products with a size of more than one million lines of code in each, an age of 8-15 years, and that were coded using C/C++. We observed six reviews of the first product, two reviews of the second, and one of the third. The reason we observed a higher number of reviews for the first product was that it had the largest number of developers working on it (four development teams). The other products had only one team each. All products were roughly equal in code size. The product with the most developers was a spearhead product with the largest number of customers that generated the majority of the revenue.

Each coding task (10-150 hours, 100-5,000 LOC) required a code review. Before each review, the responsible developer (the code author) performed smoke testing (i.e., quick, informal functional testing) to ensure that the features were working, assembled the code review material consisting of all new and modified code lines, and distributed it to the reviewers. Prior to the review meeting, the reviewers read the code and tried to find quality deviations. The company had code review checklists, but they were not actively used in the core review process and, in fact, not all employees were even aware of the checklists. The company checklist had 90 items covering 22 different areas, such as functions, argument passing, strings, memory management, conditionals, loops, design, and maintenance. The checklist items had nearly even amounts of functional and evolvability defect checks. The reviewers were mostly from the same development team working in the same code area as the author. The number of reviewers varied from 1 to 4. In some teams, only the team leader reviewed the code and, in other teams, the whole team participated.

After the review, the author fixed the identified defects based on his or her handwritten review notes. Since the company did not keep detailed records of the defects identified in code reviews, the primary author of this paper observed code review sessions. Seven out of nine review sessions lasted 30-90 minutes. Two sessions deviated from this average time: The shortest session lasted 10 minutes and the longest was 4 hours. During the review, the observer made notes regarding the identified defects and audio recorded the code review sessions for further analysis.

3.2.2 Students' Code Reviews

For this study, we also had students perform code reviews in a software quality assurance course at our university. All 87 students participated as code reviews were a mandatory part of the course. Table 1 summarizes the most important demographic variables. Programming language skills were self-evaluated on a 1-5 ordinal scale, with 1 standing for *not at all* and 5 standing for *excellent understanding*. Code review experience was determined using a 1-5 ordinal scale, with 1 representing *no code review experience* and 5 representing *participation in 10 or more code reviews*.

The students formed 23 groups with three to four students in each group. The groups were randomly divided into 11 group pairs in order to get external reviewers outside of each group and to reduce the number of review sessions that was observed by the course staff. The remaining group was paired with course personnel. Each group reviewed two Java classes, one from a member of their group and one made by a member of their pair group. See Fig. 1 for an illustration of one student group pair. Thus, 23 different classes were reviewed. The Java class under review had been implemented in a previous course exercise, but at that time, the students were unaware that it would be used for a code review. All group members had implemented the Java class, but, for the review, the Java class that had received the highest grade among the group members was used. The grading of the Java class was performed by one of the course assistants and it was based on the number of failures discovered. The students had tested the reviewed code with JUnit tests as part of a previous exercise.

The students performed the reviews individually and submitted their individual defect logs to course staff before the code review meeting. The average preparation time was 88 minutes and the average code under review was 188 LOC. The students had the possibility to use the checklist available in the course book. The checklist had 34 items, 22 for functional defects and 12 for evolvability defects. For full details of the checklist, see [14, Table 10.3]. The students were not forced to use the checklist, as this might have biased the results. However, we did not want to leave the students completely without aid, so the checklist was available. Sixty percent of the students indicated that they utilized the checklist in individual preparation, but we have no details of how thoroughly it was used. After the meeting, they submitted the meeting defect log. The students were graded

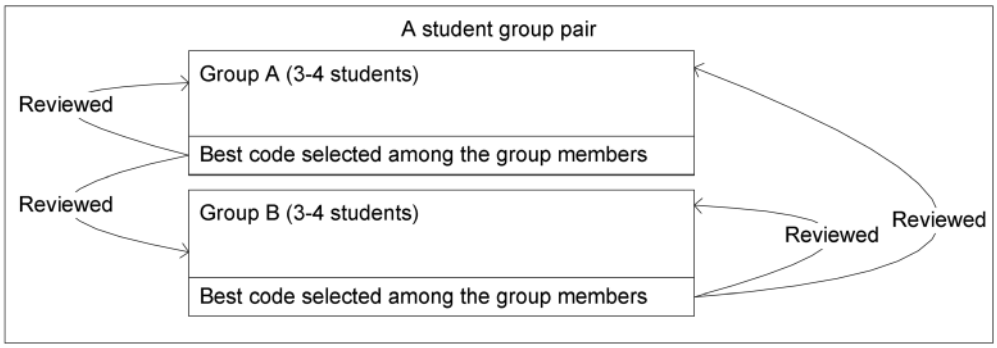

Fig. 1. Student code review arrangement.

TABLE 2
Differences in the Code Review Contexts

	Industrial Reviews	Student Reviews
LOC reviewed per session	100-5000	100-300
N of reviews	9	23
Domain	Engineering	Questionnaire data analysis
QA-method prior to code review	Functional or automated testing	JUnit testing
Individuals (reviewers)	Industrial developers	Students
Review instructions	Find problems in the code Checklists (existed but mainly not used)	Find and classify defects as functionality, evolvability, or other Checklists (60% had utilized the checklist in individual preparation)
N of reviewers	1-4 + author of the code	6-7 + author of the code
Data collection method	Observation of reviews	Submitted defect lists and source code

based on the individual defect logs, meeting activity, and the meetings' group defect logs. The defect logs were graded based on the number of defects, types of defects, and clarity of the defect reports. However, a high number of defects alone did not assure a high grade. For example, reporting several redundant, minor issues was not rewarded. The grading principles were announced before the exercise. The data for this study were gathered from the meetings' group defect logs and the reviewed code.

Table 2 summarizes the most important differentiating factors between the industrial and student reviews.

3.3 Data Analysis

Qualitative research varies greatly in research strategies and types (for details, see [52, pp. 6-7]). Based on the ideas gathered from [52], [53], and [66], the first author of this paper used a labeling process[1] (in this case, the process of conceptually labeling the defect descriptions with defect types and further refinement of the labeling) and Atlas.ti,[2] which is a program designed to support qualitative data analysis. The labels were mostly created during the analysis, rather than using a predetermined list, although our previous work [49] provided us with some ideas of what the suitable defect classes might be.

The labeling process was an individual effort carried out by the first author and it proceeded as follows: First, the industrial code review sessions from the audio tape and the notes taken during the observations were transcribed to a description of each defect. Then, defects were further analyzed and labeled. The labeling of the industrial data created roughly 80 percent of the final labels. Second, the defect logs of the students' code review sessions were analyzed, using the labels developed from the industrial data. However, some additional labels were also created as some defect types did not exist in the industrial code reviews or they had been ignored and labeled incorrectly. After this, both defect sets were rechecked to validate the quality of the labeling and to discover the need for additional labels. Throughout the process and particularly after a long labeling and defect analysis period, the labels themselves were analyzed to ensure the quality of the created labels. Based upon this, too general or too specific labels and labels with poor names were discovered. Labels that were too general and specific were typically created during the labeling process when the researcher discovered a defect for which he did not yet have a proper label. Thus, the researcher either assigned a defect to some general group or created a new label for that specific defect. There were no precise measures on what was considered too

1. The term *labeling process* is used as a substitute for *coding process*, a term that is traditionally used in qualitative data analysis because the word *code* has such a strong attachment to program code in the software engineering context.

2. For more details about the Atlas.ti program, see http://www.atlas ti.com/.

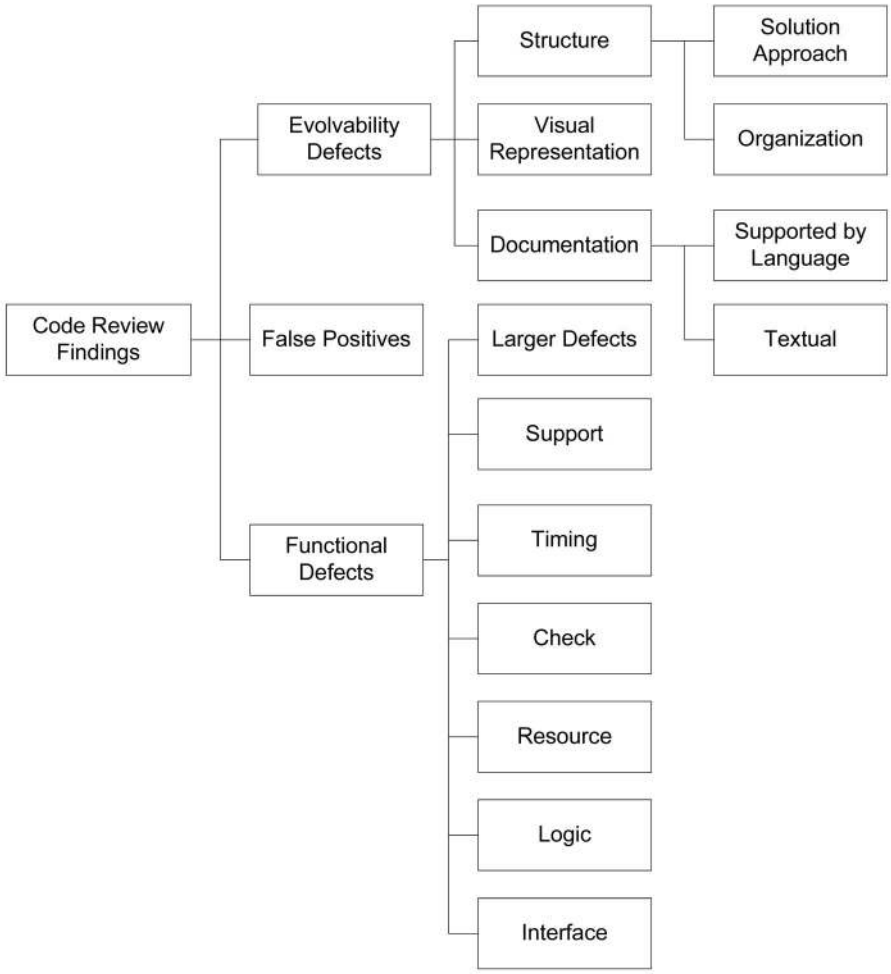

Fig. 2. Classification of the code review findings.

specific or general; the researcher tried to reach a proper level of granularity that would enable further analysis of the defects. After labeling, the student review defects were classified into larger groups that emerged from the data. This eventually resulted in the evolvability portion of the defect classification schema presented in this paper. The evolvability defect classification was created and elaborated over a longer period (i.e., several months). For the functional defects, there was no need to create a new classification, so we used existing classifications to group the functional defects.

The primary author was responsible for all data collection, analysis, and labeling. Thus, the methods of data collection and analysis are comparable and the data coding is consistent between review sessions. However, this also means that no comparison between researchers was conducted, leaving the repeatability of the created classification as an issue. Prior work [25] has shown that the classification of functional defects is generally repeatable, so it is safe to assume that our functional defect classification is reliable since it is quite similar to the one that was studied for repeatability. To address the repeatability of the evolvability defect classification, the second author studied and classified 95 defects. The agreement between raters was 82 percent, and the calculated Kappa was 0.79 indicating very good agreement. The Kappa values are similar to the ones that were obtained when studying functional defects in [25].

4 Results

This section is organized in the following way: Section 4.1 presents our overall classification of code review findings and its high-level distributions. Section 4.2 provides the details of the evolvability defect classification and its defect distributions. Similarly, Section 4.3 presents the details of the functional defect classification and distributions.

4.1 Code Review Findings

4.1.1 Classification of Code Review Findings

Fig. 2 presents an overview of our classification of code review findings, which is discussed in detail in Sections 4.2 and 4.3. This classification is based on both literature survey and empirical data. At the highest level, we have the class *code review findings*, which are divided into the following main groups: *evolvability defects*, *functional defects*, and *false positives*. These groups are taken from Siy and Votta [67]. Furthermore, the functional defect main group is subdivided into the following groups: *interface*, *logic*, *resource*, *check*, *timing*, *support*, and *larger defects*. The functional defect groups are based on several prior works (see Section 4.3 and

TABLE 3
Distribution of Code Review Findings

Main Group	Industrial reviews		Student Reviews	
Evolvability defects	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%

Table 12 for details). Finally, the evolvability defect main group is subdivided into the following groups: *documentation*, *visual representation*, and *structure*. Additionally, the documentation and structure groups have subgroups. The evolvability defect groups and subgroups are based on the data analysis discussed in this paper.

4.1.2 Distribution of Code Review Findings

Table 3 shows the distribution of code review findings arranged in the three main groups: evolvability defects, functional defects, and false positives. The classification of a defect as evolvability or functional was based on whether the defect had an impact on the functionality. If the researcher was not sure and it was not possible to ask the author of the code, a functional defect class was chosen. In both the industrial and student code reviews, over 70 percent of the findings were evolvability defects. In the industrial reviews, about 20 percent of the identified defects were functional, while in the student reviews, only 13 percent 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 discovered not to be defects either during the meeting or after. The decision whether a defect was a false positive was done by the code review team. However, for the student cases, some defects were ruled to be false positives if the researcher could not determine what the defect was based on, either the defect description or the source code. If false positives are removed, the share of evolvability defects is 77 percent in the industrial review and 85 percent in the student reviews. Thus, based upon our data, it seems that roughly four out of five of the true defects identified in the code reviews were evolvability defects.

4.2 Evolvability Defects

This section presents the details of the evolvability defect classification and the defect type distributions of the evolvability defects. We categorized the evolvability defects into three groups: documentation, visual representation, and structure. *Documentation* is information in the source code that communicates the intent of the code to humans (e.g., commenting and naming of software elements, such as variables, functions, and classes). In addition, issues that are currently embedded in and enforced by modern programming languages, whose main purpose is mostly documentation, are included in this category. For example, we see that declaring a variable to be immutable (*final* in the Java language) or limiting the scope of a method (*private* in the Java language) is mainly done for documentation (e.g., immutable variables are not modified and private methods are used from within a class only). Furthermore, people using programming languages that lack the power to limit method scope often use other means to mark private methods. For example, in PHP, developers frequently use an underscore at the beginning of a method name to indicate that it is a private method. *Visual representation* refers to defects hindering program readability for the human eye. In the present study, this subgroup contains problems mostly related to indentation and blank line usage. *Structure* indicates the source code composition eventually parsed by the compiler into a syntax tree. Structure is clearly distinguishable from documentation and visual representation, because the latter two have an impact neither on the program runtime operations or on the syntax tree generated from the source code. Table 4 shows the distribution of the evolvability defects.

In both reviews, roughly 10 percent of the evolvability defects were visual representation defects. Roughly one-third of the evolvability defects from the industrial reviews were concerned with documentation of the code. In the student reviews, almost half of the evolvability defects came from the documentation group. In the industrial reviews, 55 percent of the evolvability defects belonged to the structure group, while, in the student reviews, only 44 percent belonged to the structure group.

4.2.1 Documentation

As some documentation defects are embedded in and enforced by the programming language and others are textually documented, we have subdivided the documentation defect group into two subgroups: textual and supported by language. Table 5 shows the distribution within these two categories.

Table 6 shows the distribution of the textual documentation defects. Most of the defect type naming came from uninformative names or from violations of naming standards. Discussions with the industrial reviewers made it clear that the company strongly believed in self-descriptive naming rather than code commenting, a fact that explains

TABLE 4
Distribution of Evolvability Defects

Group	Industrial Reviews		Student Reviews	
Documentation	96	34.8%	133	46.3%
Visual representation	27	9.8%	31	10.8%
Structure	153	55.4%	123	43.9%
Total	276	100%	287	100%

TABLE 5
Distribution of Documentation Defects

Sub-Group	Industrial Reviews		Student Reviews	
Textual	65	67.7%	109	82.0%
Supported by language	31	32.3%	24	18.0%
Total	96	100%	133	100%

the difference in the distribution of naming issues. In both contexts, comment defects referred to a need to explain complicated program logic or to fix incorrect comments. It is possible that the higher amount of comment defects in the student review is explained by Java's built in the documentation system JavaDoc.

Table 7 presents the distribution of the supported by language defects. Both reviews identified such defects, although they were less common than textual defects. The only major difference was that the industrial reviewers were more eager to point out defect type immutable, a fact probably explained by corporate coding practices. Naturally, the supported by language defect distributions are affected by the programming language (e.g., with nontyped languages, element type defects are not possible).

4.2.2 Visual Representation

Table 8 shows the distribution of the visual representation defects. There were no major differences in the defect types between reviews. In general, these defects are easy to detect and there is a low risk associated with fixing them. The grouping defect, referring to the omission of blank lines to group logical code segments, is the only one that could not be fixed by automatic code formatter tools.

4.2.3 Structure

Inside the structure group, there are two subgroups: organization and solution approach. The organization subgroup consists of defects 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 of this. Solution approach defects require an alternative implementation method. For example, replacing the program's array data structure with a vector and knowing the existence of prebuilt functionality that could be used instead of a self-programmed implementation would be considered a solution approach defect. Therefore, solution approach defects are not about reorganizing existing code but rethinking the current solution and implementing it in a different way. Thus, identifying a solution approach defect is likely to require programming wit, innovative thinking, and good knowledge of the development environment and the applied development practices. Organization defects, on the other hand, can often be identified without deep knowledge of the development environment and practices, simply by assessing the code under review. Table 9 illustrates the structural defect distribution.

Some of the structure defects could actually be detected from a higher level of abstraction than the source code. For example, the need to move functionality between classes could be discovered by studying UML diagrams. Similarly, the need to create new functionality or classes could be found at the design level. Thus, the proportion of structural defect found in code reviews could be heavily affected by the quality of the architecture and design and the presence of reviews in those phases. In our data, the software company performed design reviews prior to coding, but the industrial developers still found more design level defects

TABLE 6
Distribution of Textual Defects

Defect Type	Industrial Reviews		Student Reviews	
Naming	45	69.2 %	30	27.5%
Comments	16	24.6 %	65	59.6%
Debug info	1	1.5 %	13	11.9%
Others	3	4.6 %	1	0.9%
Total	65	100%	109	100%

TABLE 7
Distribution of Supported by Language Defects

Defect Type	Industrial Reviews		Student Reviews	
Element type	11	35.5%	13	54.2%
Immutable	13	41.9%	2	8.3%
Visibility	5	16.1%	6	25.0%
Void parameter	2	6.5%	0	0%
Element reference	0	0	3	12.5%
Total	31	100%	24	100%

TABLE 8
Distribution of Visual Representation Defects

Defect Type	Industrial Reviews		Student Reviews	
Bracket usage	5	18.5%	11	35.5%
Indentation	6	22.2%	5	16.1%
Blank line usage	8	29.6%	7	22.6%
Space usage	4	14.8%	0	0%
Grouping	2	7.4%	3	9.7%
Long line	2	7.4%	5	16.1%
Total	27	100%	31	100%

TABLE 9
Distribution of Structure Defects

Sub-group	Industrial Reviews		Student Reviews	
Organization	73	47.7%	85	69.1%
Solution approach	80	52.3%	38	30.9%
Total	153	100%	123	100%

TABLE 10
Distribution of Organization Defects

Defect Type	Industrial Reviews		Student Reviews	
Move functionality	17	23.3 %	4	4.7%
Long subroutine	9	12.3 %	14	16.5%
Dead code	21	28.8 %	15	17.7%
Duplication	11	15.1 %	14	16.5%
Complex code	3	4.1 %	8	9.4%
Statement issues	2	2.7 %	13	15.3%
Consistency	3	4.1 %	1	1.2%
Others	7	9.6 %	16	18.8%
Total	73	100%	85	100%

than students who were provided with design by the course staff. In fact, this was the biggest overall difference between industrial and student reviews. For example, Table 10 presents the most frequent organization defects, showing that industrial reviewers identified more move functionality defects, whereas the students identified more statement issues. This trend was also noticeable inside *dead code* defects where the students identified more low-level defects, such as unnecessary headers and unnecessary variables, while the industrial reviewers found more uncalled functions and branches of code that were never executed. Additionally, in Table 11, the distribution of the *solution approach* defect types is presented. This data shows that industrial reviews occasionally found a need for new functionality (e.g., new classes) to keep the software evolvable, but such issues were not identified in the student reviews. However, this result is not surprising because the industrial software system was much larger and more complex than the students' exercise application. Unfortunately, we cannot assess the effectiveness of the industrial design reviews in relation to structure defects detectable at design level as such detailed data were not available.

The solution approach subgroup contains defects ranging from simple function call changes to a complete rethinking of the current implementation. Interestingly, we found semantic versions of dead code and duplication (see Table 11). *Semantic dead code* is code that is executed in the program but serves no meaningful purpose. Similarly, *semantic duplication* means that a program contains unnecessary duplication of syntactically no-equal code blocks with equal intent (e.g., different sorting algorithms, such as quicksort and heapsort, have equal intent but are not syntactically equal). The defect type *others* contains a wide range of defects. The issues in this group represent the solution approach defect category in its most fruitful form. When attempting to spot solution approach issues, books, guides, and education provide little help. Instead, the reviewer's skills and experience are crucial in detecting these defects. Examples include using arrays instead of other more complex structures, enabling the easier removal of several data items from the database, and simpler ways of performing computing and comparison operations.

4.3 Functional Defects

4.3.1 Functional Defect Classification

To address the shortcomings of the existing functional defect classification mentioned in Section 2.2, we created a functional defect taxonomy based on our data and existing classifications. However, our classification of functional defects should be seen as a superset of the previous classifications rather than a completely new one. This classification has seven groups: resource, check, interface, logic, timing, support, and larger defects. Table 12 shows how the categories are based on prior work and enables comparison between our data sets and previous work.

Resource defects refer to mistakes made with data, variables, or other resource initialization, manipulation, and release. We make no distinction between data being stored in variables, other types of data structures, or memory for the data being allocated statically or dynamically. Furthermore, it makes no sense to separate memory allocation issues from similar defects made using other resources, such as files or database handles. Our decision to combine assignment and data defects is supported by El Emam and Wieczorek [25], as they indicated that reviewers often have problems distinguishing these categories from each other.

Check defects are validation mistakes or mistakes made when detecting an invalid value. Interface defects are mistakes made when interacting with other parts of the software, such as an existing code library, a hardware device, a database, or an operating system. The group logic contains defects made with comparison operations, control flow, and computations and other types of logical mistakes. The timing category contains defects that are possible only in multithread applications where concurrently executing threads or processes use shared resources. Support defects relate to support systems and libraries or their configurations. For

TABLE 11
Distribution of Solution Approach Defects

Defect Type	Industrial Reviews		Student Reviews	
Semantic duplication	8	10.0%	7	18.4%
Semantic dead code	10	12.5%	4	10.5%
Change function	25	31.3%	0	0%
Use standard method	16	20.0%	19	50.0%
Create new functionality	6	7.5%	0	0%
Others	9	11.3%	8	21.1%
Minor	6	7.5%	0	0%
Total	80	100%	38	100%

TABLE 12
Comparison of Functional Defect Classifications

Our Classification	Resource	Check	Interface	Logic	Timing	Support	Larger defects
Basili and Selby [7]	Initialization, Data	-	Interface	Control, Computation	-	-	-
Chillarege et al. [18]	Assignment	Checking	Interface	Algorithm	Timing/ serialization	Build/ package/ merge	Function
Humphrey [32]	Assignment, Data	Checking	Interface	Function	System	Packaging, Environment	
Beizer [10]	Data	-	Integration, System and software architecture	Structural bugs	Timing	-	Functionality as implemented
Kaner et al. [37]	Initial and later states, Handling data, Load conditions	Error handling	Hardware	Calculation, Control flow, Boundary-errors	Race conditions	Source and version control	User interface errors
IEEE [33]	Data handling	-	Interface/timing	Logic	Interface/ timing	Data	-
Grady [31]	Data Handling, Data Definition	Error Checking	Module Interface	Logic, Logical Description	Process Communications	-	-

TABLE 13
Excluded Defect Categories

Basili and Selby [7]	**Chillarege et al.** [18]	**Humphrey** [32]	**Beizer**[10]	**Kaner et al.** [37]	**IEEE** [33]	**Grady** [31]
Cosmetic	Documentation	Syntax, Documentation	Functional requirements, Test definition or execution bugs.	Testing errors, Documentation	Documentation, Documentation quality, Enhancement	Standards, Module Design

example, version control and build-management system configurations can introduce defects. Similarly, using the wrong version of an external library can cause defects. Larger defects, unlike those presented above, cannot be pinpointed to a single, small set of code lines. Larger defects typically refer to situations in which functionality is missing or implemented incorrectly and such defects often require additional code or larger modifications to the existing solution.

Table 13 shows the defect categories from prior works that we have excluded. We left out four types of defects. First, defect categories that contained syntax errors have been excluded because they are best identified by the compiler (syntax by Humphrey [32]). Second, we left out categories measuring the impact of the defects, as we think that defect impact and technical type should be measured on different scales (cosmetic by Basili and Selby [7]). Third, defect categories whose content belong to the evolvability defects have been excluded (documentation by Chillarege et al. [18], Kaner et al. [37], IEEE, and Humphrey [32], enhancement by IEEE, and Standards by Grady). Fourth, defects that are not defects in the code but reside in the

TABLE 14
Distribution of Resource Defects

Defect Type	Industrial Reviews		Student Reviews	
Variable initialization	10	43.5%	0	0%
Memory management	7	30.4%	0	0%
Data and resource manipulation	6	26.1%	1	100%
Total	23	100%	1	100%

requirements, design, or in the test definitions have been excluded (functional requirements and test definition or execution bugs by Beizer, testing errors by Kaner et al., module design by Grady).

4.3.2 Distribution of Functional Defects

Tables 14, 15, 16, 17, and 18 show the types and distributions of the resource, check, interface, logic, and larger defects that were identified. We were not able to identify any timing or support defects in the reviews.

As we can see, resource defects were mostly present in the industrial reviews. There are two reasons for this. First, the students' application was small and did not involve heavy data manipulation. Second, the language used, Java, handles or enforces many issues in resource management, making such defects impossible.

Two main types were present in the check defects category: the need to check function return values or the need to check variable values. The interface defects were missing or incorrect function calls or parameters and they were mostly present in industrial reviews. In the logic defects category, there were no major differences between the review contexts. The high number of compute defects was caused by the fact that the students' application required several computations.

We also identified three types of larger defects that could not be pinpointed to a limited set of code lines. No larger defects were found in the student reviews, likely due to the small application size. For example, in the industrial reviews, there were cases where a feature was not complete and, thus, would not work in all scenarios, issues related to the user interface, or defects existing in parts of the application code that were not under review.

5 Discussion

In this section, we answer the research questions and then compare our results with related work. Section 5.3 describes the implications of the results. In Section 5.4, we suggest how the review responsibilities may be assigned based on the identified defect classes. Finally, in Section 5.5, we discuss the limitations of the study and evaluate our work.

5.1 RQ1: Distribution between Functional and Evolvability Defects

Research question 1 was given as follows: *What is the distribution between functional defects and evolvability defects?* Based on our study, roughly 75 percent of the findings identified in the code reviews were evolvability defects that would not cause runtime failures. This research question was confirmatory, as Siy and Votta [67] had previously proposed, and based on data from a single company that most code review findings are evolvability defects. Our results confirm their findings.

Comparing our results (see Table 19) with the numbers of Siy and Votta indicates that our study has a slightly higher proportion of evolvability defects, a slightly lower proportion of false positives, and a similar proportion of functional defects. 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 defects may have decreased since the author might have disregarded some defects, considering them to be false positives.

To determine whether the large number of evolvability defects (as 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 [18], [25], [58] 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 [32] and Runeson and Wohlin [63] as they had high shares of syntactical errors because they performed the reviews before compilation.

TABLE 15
Distribution of Check Defects

Defect Type	Industrial Reviews		Student Reviews	
Check function	12	52.2%	13	46.4%
Check variable	9	39.1%	14	50.0%
Check user input	2	8.7%	1	3.6%
Total	23	100%	28	100%

TABLE 16
Distribution of Interface Defects

Defect Type	Industrial Reviews		Student Reviews	
Function call	9	64.3%	1	100.0%
Parameter	5	35.7%	0	0%
Total	14	100%	1	100.0%

TABLE 17
Distribution of Logic Defects

Defect Type	Industrial Reviews		Student Reviews	
Compare	3	25.0%	4	21.1%
Compute	0	0%	3	15.8%
Wrong location	3	25.0%	2	10.5%
Algorithm/performance	5	41.7%	4	21.1%
Other	1	8.3%	6	31.6%
Total	12	100%	19	100%

Fig. 3 compares eight data sets, showing the proportions of functional and evolvability defects. In five of the eight data sets, the majority of the defects are evolvability defects, and in seven of the sets, over half of the defects are evolvability defects. Based on the figure, it seems likely that the majority of the defects detected in code reviews are, in fact, evolvability defects. 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 amount and types of defects detected. Unfortunately, such information is not available in studies *Si*, *O*, *E1*, *E2*, and *C*. In study *So*, 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 personally tested the application and were either personally responsible (industry case) or had been graded based on the number of faults (student case). Thus, it is likely that the higher amount 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 defects and, as previously discussed in Section 2.3, it is possible that such evolvability defects were ignored or categorized as functional defects. Furthermore, in study *C*, there appeared to be a defect class that possibly contained both functional and evolvability defects. Additionally, in those studies, the authors did not make a distinction between functional and evolvability defects. 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 probable that, in those studies, the proportions of evolvability defects 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, strictness of the code review process, and the company culture. Because of these shortcomings, Fig. 3 should be studied with caution.

TABLE 18
Distribution of Larger Defects

Defect Type	Industrial Reviews		Student Reviews	
Completeness	4	28.6%	0	0.0%
GUI	5	35.7%	0	0.0%
Check outside code	2	14.3%	0	0.0%
Total	11	100%	0	0.0%

5.2 RQ2: Defect Types and Defect Classification

Research question 2 was given as follows: *What different types of evolvability and functional defects are found in code reviews, and how can they be classified based on the defects' technical content?* The classification scheme is presented in Fig. 2.

This research question was both exploratory and confirmatory. For the functional defects, we were able to confirm that the existing classifications match well with each other and that such defects are found during code review. For evolvability defects, we created a new classification.

The evolvability classification also appears to be generally repeatable, as the measured Kappa values indicated very good agreement between the raters. We recognize that the defect classification is difficult [7], [10], that there are borderline cases for which it is difficult to classify the defect, and that different people make different interpretations. However, this issue exists in prior defect classifications as well. The classification also appears to be exhaustive in the study context, as we were able to classify all of the defects in our data set. However, it is possible that the classification is not exhaustive with other data sets.

Next, we compared our evolvability defect classification scheme to prior work. Textual documentation defects have been reported in previous studies and we found such defects in six of the seven code review data sets shown in Fig. 3 (excluding study *O*). The comparison shows a large fluctuation between textual defect proportions: from 17 percent up to 50 percent. Further analysis of textual defects shows fluctuation between the two largest textual defect types, comment and naming. El Emam and Wieczorek [25] found hardly any naming defects, but studies [54], [72] of source code modifications confirm that code element renaming is very frequently performed in practice. It is possible that naming issues have simply been ignored in some reviews.

For visual representation, we found that such defects have only been reported by Siy and Votta. Our study had proportions of 10 percent and 11 percent, while Siy and Votta had a proportion of 12 percent. Thus, based on limited evidence, it appears that visual representation accounts for approximately 10 percent of the evolvability defects detected in code reviews.

TABLE 19
Comparing Our Results to the Results of Siy and Votta

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

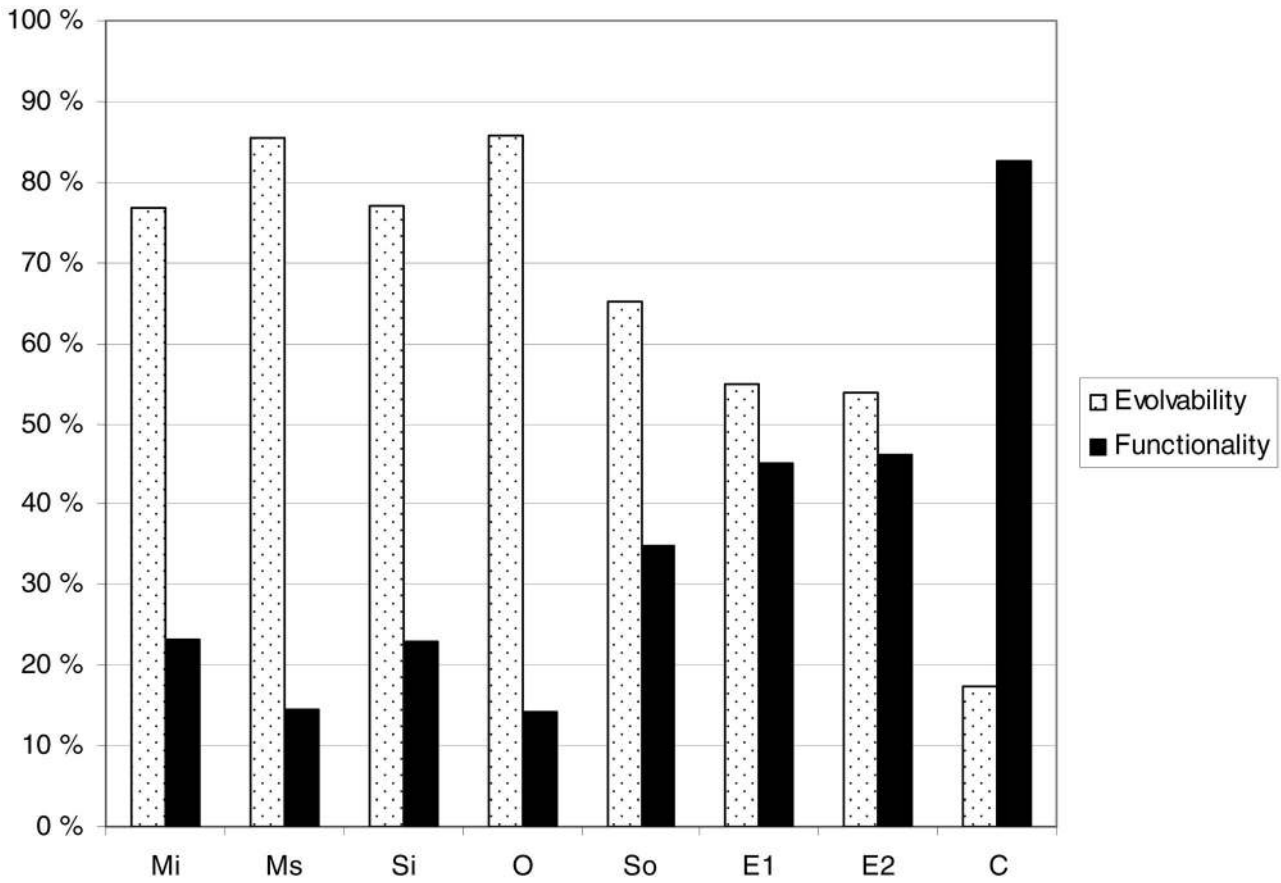

Fig. 3. Defect distribution between functionality and evolvability. Mi is the industrial data, Ms is the student data, Si [67], O [58], So [68], E1 and E2 [25], and C [16], [18]

In the structure group, we have two categories: organization and solution approach. Fixing the organization subgroup defects is known as *refactoring* and it has been studied extensively in the past [50]. 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 percent to 55 percent. For the solution approach defects, we are not aware of any research presenting or discussing such defects in the software engineering domain. For structural defects, we noticed that industrial developers found more defects that could have been detected from design documents than did student reviewers. The industry case had design reviews, but we have no data of the detected defects from that stage, so we cannot evaluate the effects of design and architectural reviews (e.g., [19], [38], [59]) that are related to the amount of structure defects. Thus, it seems likely that the industrial software had more high-level defects due to its size and complexity. Furthermore, little empirical research is available on the defect types detected in architectural or design reviews. The only source we found was by Chillarege et al. [18] who, as expected, indicated that more high-level defects are found in design reviews than in code reviews.

Our functional defect classification was heavily based on existing work and, in our data set, we found functional defects belonging to five of the seven function defect classes presented. We did not find defects belonging to the timing group because the review code did not have multiple threads and support defects were not found since it is very unlikely that a reviewer will detect such defects when reviewing a code. Based on our study, we can confirm that existing function classifications are solid and that they work well with code review defect data. We compared our functional defect proportions to those of prior studies, but the comparison did not reveal any interesting consistencies or trends.

5.3 Implications

5.3.1 Implications to Practitioners

Having discussed the value of evolvability defects, we can outline some practical implications:

- Code reviews seem particularly valuable for software product or service businesses in which the same software or service is modified and extended over time. 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 business should consider targeting reviews for those modules that will be modified in the future.
- Only 10 percent of the evolvability defects belong to the visual representation group that can be (mostly) automatically detected and fixed. Organizations should consider using pretty printers for automatically fixing visual representation defects so that code reviews can focus on issues that are more important. The majority of the evolvability defects cannot be fixed simply by using tools. However, tools can help in detecting structural defects, although human intervention is still required for analyses and repair.
- An established and empirically grounded defect classification should be used as a baseline when building checklists for code reviews or when creating coding standards for the developers. This assures that most typical defect classes are covered.
- An organization should try to link defect severity and fixing time to technical type since this may allow savings. The most costly mistakes could be prevented or found earlier in the development process.
- Classifying defects based on technical content can help organizational and individual learning. A classification also provides information of the applied quality practices and enables comparisons between them.
- If organizations are using the classification system to classify defects (instead of building checklists or instructions), it may be beneficial to start with the three main classes: visual representation, documentation, and structure. Furthermore, the functional defect types that are most useful for the organization should be considered. It is easy to extend the classification later based on the organizational needs, but starting with a complex classification can lead to resistance with minimal additional benefits.

5.3.2 Implications for Research

The results relate to prior studies in which code reviews have been compared with software testing [7], [36], [42], [55], [71]. In these studies, only functional defects causing failures have been counted in order to make a fair comparison between testing and code review. The studies show no consistent differences in defect detection effectiveness between review and testing. Assuming that, in these studies, the ratio of evolvability defects to functional defects has ranged from 3:1 to 5:1, we can suggest that code review is superior as it not only finds the same amount of functional defects as testing but also identifies a large number of evolvability defects. However, it is not proven that the ratios of 3:1 to 5:1 can be transferred to earlier studies and, therefore, this result needs further research before it can be generally accepted.

Further, the discovered evolvability defects can be used as a basis for further research when creating tools that can detect various evolvability and functional defects. However, creating tools that recognize possibilities of using an alternative approach in code implementation or that recognize semantic duplication will likely be challenging.

5.3.3 Implications to Scenario-Based Reviews

In the context of requirement document inspections, previous studies have proposed scenario-based reading [6], [60]. The scenarios are based on either perspectives reflecting the organizational roles (e.g., requirement engineer or test case designer) [6], [48], [61] or defect classes [60]. We see that, in the context of code review, defect classes might be more suitable than perspectives as many of the organizational roles do not work directly with the code.

Five reviewers may utilize the following roles based on our classification. The documentation reviewer should approach the code from a textual point of view and try to determine whether it contains enough information in names, comments, and code element specification. The code structure reviewer should focus on structural organization defects. Suitable people for this role are module-level architects, as they have suitable knowledge of how the module should be organized. The solution approach reviewer may be a suitable role for experienced guru developers, who have a wide knowledge of the system and are able to think of alternative implementation approaches. The resource and interface reviewer should concentrate on functional defects belonging to the resource and interface categories. Finally, the check and logic reviewer should focus on functional defects belonging to the check and logic categories, requiring an alert individual who has the patience to look for logical errors and missing checks.

Defects belonging to the visual representation group are best handled with tools, so we assigned no role for this defect group. When presenting the above idea for the case company, one employee suggested that only the last three roles are needed, as the defects detected by the first two roles should come automatically when reviewing and trying to understand the code.

5.3.4 Value of Defect Classifications

One may question the value of technical defect classifications as there is currently no research (that we are aware of) that would have linked technical defect type and defect severity or defect fixing time. Overall, there has been little work linking defect severity to other software attributes. Currently, the only work of this nature that we aware of is by Zhou and Leanung [73], who tried to link defect severity with code metrics. However, there is anecdotal evidence that a link between defect technical type and severity or fixing time could exist. For example, our discussions with industrial developers indicate that memory leaks can cause

severe problems that are difficult to fix. Unfortunately, we have no hard data on this issue. Furthermore, the link between defect type and severity would probably be context dependent. Nevertheless, studies linking technical defect type to defect severity or fixing time could help practitioners avoid the most severe and time-consuming defects. Even though the link between defect technical type and defect severity or fixing time has not yet been established, defect classifications are important because they allow comparisons between quality assurance methods. They are also important because they can be used to focus code reviews, build and organize checklists or coding standards, and can make it easier for individuals or organizations to learn from typical mistakes.

5.4 Limitations

To assess the limitations of this work, we studied the possible weaknesses in qualitative studies and analyzed internal and external validities based on [15], [20], [52].

5.4.1 Evaluation and Threats to Internal Validity

The limitations of the study stem from the difficulty of classifying defects, the repeatability of defect classification, and the issue of researcher bias stemming from personal assumptions and values.

The data are retained, but only the student data may be viewed by other researchers due to the case company agreements. It should be noted that the effect of researcher bias could have been controlled during the data coding had there been several researchers. Thus, triangulation of the researchers could have benefited the data analysis.

It could be argued that having several data sources is a weakness since the data sources are heterogeneous and not comparable. However, our data sources are not different from those in any two studies where one is performed in a real industrial context and the other is performed with students. Furthermore, having two data sources is an improvement over past studies that have arrived at their conclusions mostly based on data from only a single source (e.g., an experiment or data from a single company).

Method triangulation was also performed. We observed the industrial code review sessions but used the defect logs and the source code in the student code reviews. However, the method triangulation was not done within a data source, as the same method was applied to all reviews. This may have created a slight bias between data sources.

It is possible that the presence of the researcher in the industrial reviews and the fact that the student review sessions and review logs were graded could have affected the results. Students might have tried to find as many defects as possible, leading to higher counts of evolvability defects. On the other hand, it would be difficult to assume that such behavior would have occurred when observing the industrial reviews.

Grading of the students might have introduced some bias, but the absence of grading could have resulted in lack of motivation, which could have introduced additional bias. It is possible that students cheated in their individual defect lists, although it is unlikely. It would be easy to identify cheating if several members of the team provided similar defect lists. During the grading of the personal defect lists performed by the first author, there was no cheating detected. Thus, there is no reason to believe that the students had not reviewed the code before the meeting. Cheating during the meeting was controlled by the course assistants, who were present during the review meetings.

As mentioned in Section 3.2.2, all students had implemented the reviewed Java class. It is unclear how this affects the results. It is possible that, as the students were all aware of the functionality, they focused more on the evolvability issues. On the other hand, it is possible that the students' prior knowledge made them more aware of the possible sources of functional defects.

The impact of an individual evolvability defect is also unknown. Thus, it is possible that a fix to solution approach defect will turn out to be a better or worse solution in the future. In this study, it was not possible to study the impact of individual evolvability defect in a longitudinal fashion. Thus, it is likely that, in the future, some evolvability defects would turn out to be false positive, but there is no way to estimate the amount of such defects.

5.4.2 Evaluation and Threats to External Validity

It must be noted that, in the industrial reviews, the preparation time fluctuated and, in some cases, the reviewers admitted that they had not properly prepared for the review. These kinds of problems in industrial software reviews are also reported by Laitenberger [43]. Thus, in the presence of variations in the review process, one may question the generalizability of our defect distribution results. First, variations in the real-world code reviews are natural and are likely to occur in most organizations, as previous work has also shown [43]. Second, it is unclear how these variations affect the defect types. It is also possible that the variations canceled each other out, leaving little overall variation. We must also note that poor preparation was not an issue in the student reviews, where preparation time was an average of 88 minutes and the code under review was rather short, an average of 188 lines.

One issue affecting generalizability is the context in which the code reviews were performed. The number and type of defects detected with any quality assurance method are largely affected by other factors not directly related to the quality assurance method. These factors include the development environment, the individual's skills, and the prior quality assurance methods used. It is arguable that having a great share of evolvability defects in this study is because unit-level functional testing was performed before the code reviews. Naturally, the number of functional defects detected would have been greater if there had not been any functional testing prior to code review. However, we think that this study represents a realistic view, as in our experience, code reviews are often performed after initial testing in industry. The student reviews were performed after automated unit testing. It would seem awkward to perform team-level code review before creation of unit tests when there are methodologies advocating that unit tests should be created even before the code is written [8]. The industrial reviews were performed after the developer had completed the task. In the industrial context, where the developer added features to a complex application, it was

natural for the developer to code part of the task, then quickly test it before coding another part of the task. Performing code reviews on untested code was not an option in the industrial setting because the developer could not complete the task without knowing whether the code was working. In addition, organizing a code review in the middle of the task would have been very laborious and difficult to organize.

Several code review sessions were studied but in only two contexts. However, there were also differences within each context. In the industrial code reviews, we studied sessions from three different departments that each had slightly different code review practices. In the student reviews, there were 23 student groups (roughly 80 individuals) with a single individual participating only in one review session. Although using two different data sources offers limited generalizability, it is an improvement over prior work, as previous studies have mostly used a single data set or used two data sets but from a single source.

6 Conclusion

This paper has made two contributions to the software engineering community. First, it affirms that code reviews are a good tool for detecting code evolvability defects that cannot be found in later phases of testing because they do not affect the software's visible functionality. In this study, the ratio of evolvability defects to functional defects ranged between 5:1 and 3:1. Our results hold for the most realistic scenario, one in which a developer has performed a quick functional test before submitting the code for review.

Second, this paper has provided the most comprehensive qualitative classification of code review findings to date. For evolvability defects, we created a new classification containing three main categories: documentation, visual representation, and structure. For functional defects, we reviewed several existing classifications and, based on these classifications, we extracted seven groups: resource, check, interface, logic, timing, support, and larger defects. This classification will be useful in the following situations. First, it could be used for assigning roles in a code review team using defect-based reading. Using such roles will increase commitment through responsibility, as only a single individual is responsible for particular viewpoints. Second, the classification could be used when creating or organizing code review checklists and coding standards as it would assure that most defect types are covered. Third, the evolvability part of the classification could act as a starting point when assessing system evolvability. The classification allows us to state the nature of the evolvability problems that can indicate the required actions to fix and prevent the problems in the future. For example, evolvability problems stemming from documentation defects can have an equal impact as the structural defects, but fixing documentation defects is likely to be cheaper and likely to require different actions than fixing structural defects. Fourth, understanding the defect types and defect classification will be useful for people who are creating defect detectors, code metrics tools, or code maintainability issue detectors or for those developing new programming languages.

We propose four areas for future work. First, to strengthen the empirical knowledge in this area, it would be beneficial for future researchers to study the distributions of defects identified in different settings. As previously discussed, one study [11] found that only 17 percent of faults found in document inspections would eventually be experienced by the customer or detected in testing. Thus, studies of defect types detected in other reviews could benefit the software engineering community. Second, further studies of the effects of the evolvability defects are required. We found only one study [69] comparing the impact of code structure against code documentation. When companies work with limited resources, it would be in their interest to know which evolvability issues have the highest impact on code comprehension and future development effort. Third, the reviewer's experience, training, and personal taste are likely to affect the detected evolvability issues. For a junior developer, for example, it can even be impossible to detect semantic duplication if he or she does not know the application thoroughly. Such studies would clarify when capture-recapture methods [24] could be used to estimate the evolvability defect content and when the differences in the reviewers' backgrounds make it impossible. Finally, defect types and distributions of defects found by other quality assurance methods, such as unit testing and acceptance testing, merit further study. Having a broad knowledge of the defect types detected by different quality assurance methods would help software engineering practitioners choose the right tools for their quality assurance toolbox.

Acknowledgments

This research has been supported by the Finnish Funding Agency for Technology and Innovation (TEKES) and by the Finnish Graduate School on Software and Systems Engineering (SOSE). The authors would like to thank Juha Itkonen, Jari Vanhanen, Kristian Rautiainen, Jarno Vähäniitty, Maria Paasivaara, and Arttu Piri for their valuable contributions to the research presented in this paper. The authors would also like to thank the other members of their research group who improved this paper through their internal paper review meetings. The authors express their gratitude toward the anonymous reviewers and the associate editor, who provided many helpful suggestions for improving this paper. This research would have not been possible without the case company, which prefers to remain anonymous, and the students participating in the Software Testing course.

References

[1] AFOTEC, *Software Maintainability Evaluation Guide.* Dept. of the Air Force, HQ Air Force Operational Test and Evaluation Center, 1996.

[2] R.S. Arnold, "Software Restructuring," *Proc. IEE,* vol. 77, no. 4, pp. 607-617, 1989.

[3] A. Aurum, H. Petersson, and C. Wohlin, "State-of-the-Art: Software Inspections after 25 Years," *Software Testing, Verification, and Reliability,* vol. 12, no. 3, pp. 133-154, 2002.

[4] 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, pp. 77-87, Jan. 2003.

[5] R.D. Banker, S.M. Datar, C.F. Kemerer, and D. Zweig, "Software Complexity and Maintenance Costs," *Comm. ACM,* vol. 36, no. 11, pp. 81-94, 1993.

[6] V.R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sørumgård, and M.V. Zelkowitz, "The Empirical Investigation of Perspective-Based Reading," *Empirical Software Eng.,* vol. 1, no. 2, pp. 133-164, 1996.

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

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

[9] K. Beck, *Extreme Programming Explained.* Addison-Wesley, 2000.

[10] B. Beizer, *Software Testing Techniques.* Van Nostrand Reinhold, 1990.

[11] T. Berling and T. Thelin, "An Industrial Case Study of the Verification and Validation Activities," *Proc. Ninth Int'l Software Metrics Symp.,* pp. 226-238, 2003.

[12] B. Boehm and V.R. Basili, "Top 10 List [Software Development]," *Computer,* vol. 34, no. 1, pp. 135-137, Jan. 2001.

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

[14] I. Burnstein, *Practical Software Testing.* Springer, 2002.

[15] D.T. Campbell and J.C. Stanley, *Experimental and Quasi-Experimental Design for Research.* Rand McNally College, 1966.

[16] J.K. Chaar, M.J. Halliday, I.S. Bhandari, and R. Chillarege, "In-Process Evaluation for Software Inspection and Test," *IEEE Trans. Software Eng.,* vol. 19, no. 11, pp. 1055-1070, Nov. 1993.

[17] S.R. Chidamber, D.P. Darcy, and C.F. Kemerer, "Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis," *IEEE Trans. Software Eng.,* vol. 24, no. 8, pp. 629-639, Aug. 1998.

[18] 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," *IEEE Trans. Software Eng.,* vol. 18, no. 11, pp. 943-956, Nov. 1992.

[19] P. Clements, R. Kazman, and M. Klein, *Evaluating Software Architectures: Methods and Case Studies.* Addison-Wesley, 2002.

[20] T.D. Cook and D.T. Campbell, *Quasi-Experimentation: Design and Analysis Issues for Field Settings.* Rand McNally College, 1979.

[21] W. Cunningham, "The WyCash Portfolio Management System," *Proc. Seventh Ann. Conf. Object-Oriented Programming Systems, Languages, and Applications,* pp. 29-30 (addendum), 1992.

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

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

[24] K. El Emam and O. Laitenberger, "A Comprehensive Evaluation of Capture-Recapture Models for Estimating Software Defect Content," *IEEE Trans. Software Eng.,* vol. 26, no. 6, pp. 518-540, June 2000.

[25] K. El Emam and I. Wieczorek, "The Repeatability of Code Defect Classifications," *Proc. Ninth Int'l Symp. Software Reliability Eng.,* pp. 322-333, 1998.

[26] M.E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," *IBM System J.,* vol. 15, no. 4, pp. 182-211, 1976.

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

[28] D.A. Garvin, "What Does 'Product Quality' Really Mean?" *Sloan Management Rev.,* vol. 26, no. 1, pp. 25-43, Fall 1984.

[29] T. Gilb and D. Graham, *Software Inspection.* Addison-Wesley, 1993.

[30] N. Gorla, A.C. Benander, and B.A. Benander, "Debugging Effort Estimation Using Software Metrics," *IEEE Trans. Software Eng.,* vol. 16, no. 2, pp. 223-231, Feb. 1990.

[31] R.B. Grady, *Practical Software Metrics for Project Management and Process Improvement.* Prentice Hall, 1992.

[32] W.S. Humphrey, *A Discipline for Software Engineering.* Addison-Wesley Longman, 1995.

[33] IEEE, *IEEE Standard Classification for Software Anomalies,* IEEE Std. 1044-1993, 1994.

[34] *IEEE Standard Glossary of Software Engineering Terminology.* IEEE, 1990.

[35] S. Jamieson, "Likert Scales: How to (Ab)Use Them," *Medical Education,* vol. 38, no. 12, pp. 1217-1218, Dec. 2004.

[36] E. Kamsties and C.M. Lott, "An Empirical Evaluation of Three Defect-Detection Techniques," *Proc. Fifth European Software Eng. Conf.,* pp. 362-383, 1996.

[37] C. Kaner, J. Falk, and H.Q. Nguyen, *Testing Computer Software.* John Wiley & Sons, 1999.

[38] R. Kazman, M. Klein, and P. Clements, "ATAM: Method for Architecture Evaluation," Technical Report CMU/SEI-2000-TR-004, 08/2000, 2000.

[39] B.A. Kitchenham and S.L. Pfleeger, "Software Quality: The Elusive Target," *IEEE Software,* vol. 13, no. 1, pp. 12-21, 1996.

[40] B.A. Kitchenham, S.L. Pfleeger, L.M. Pickard, P.W. Jones, D.C. Hoaglin, K. El Emam, and J. Rosenberg, "Preliminary Guidelines for Empirical Research in Software Engineering," *IEEE Trans. Software Eng.,* vol. 28, no. 8, pp. 721-734, Aug. 2002.

[41] T.R. Knapp, "Treating Ordinal Scales as Interval Scales: An Attempt to Resolve the Controversy," *Nursing Research,* vol. 39, no. 2, pp. 121-123, Mar.-Apr. 1990.

[42] O. Laitenberger, "Studying the Effects of Code Inspection and Structural Testing on Software Quality," *Proc. Ninth Int'l Symp. Software Reliability Eng.,* pp. 237-246, Nov. 1998.

[43] O. Laitenberger, M. Leszak, D. Stoll, and K. El Emam, "Quantitative Modeling of Software Reviews in an Industrial Setting," *Proc. Sixth Int'l Software Metrics Symp.,* pp. 312-322, 1999.

[44] O. Laitenberger and J. DeBaud, "An Encompassing Life Cycle Centric Survey of Software Inspection," *J. Systems and Software,* vol. 50, no. 1, pp. 5-31, 2000.

[45] T.D. LaToza, G. Venolia, and R. DeLine, "Maintaining Mental Models: A Study of Developer Work Habits," *Proc. 28th Int'l Conf. Software Eng.,* pp. 492-501, 2006.

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

[47] C. Linnaeus and J.F. Gmelin, *Systema Naturae per Regna Tria Naturae, Secundum Classes, Ordines, Genera, Species, cum Characteribus, Differentiis, Synonymis, Locis.* Laurentius Salvius, 1758.

[48] J.C. Maldonado, J. Carver, F. Shull, S. Fabbri, E. Dória, L. Martimiano, M. Mendonça, and V. Basili, "Perspective-Based Reading: A Replicated Experiment Focused on Individual Reviewer Effectiveness," *Empirical Software Eng.,* vol. 11, no. 1, pp. 119-142, 2006.

[49] M.V. Mäntylä and C. Lassenius, "Drivers for Software Refactoring Decisions," *Proc. Int'l Symp. Empirical Software Eng.,* pp. 297-306, 2006.

[50] T. Mens and T. Tourwe, "A Survey of Software Refactoring," *IEEE Trans. Software Eng.,* vol. 30, no. 2, pp. 126-139, Feb. 2004.

[51] R.J. Miara, J.A. Musselman, J.A. Navarro, and B. Shneiderman, "Program Indentation and Comprehensibility," *Comm. ACM,* vol. 26, no. 11, pp. 861-867, 1983.

[52] M.B. Miles and M.A. Huberman, *Qualitative Data Analysis.* Sage Publications, 1994.

[53] T. Moilanen and S. Roponen, *Kvalitatiivisen Aineiston Analyysi Atlas.Ti-Ohjelman Avulla ("Analyzing Qualitative Data with Atlas.Ti Software).* Kuluttajatutkimuskeskus, 1994.

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

[55] G.J. Myers, "A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections," *Comm. ACM,* vol. 21, no. 9, pp. 760-768, 1978.

[56] I. Niiniluoto, "Käsitetyypit Ja Mittaaminen ("Concept Types and Measurement")," *Johdatus Tieteenfilosofiaan: Käsitteen Ja Teorianmuodostus ("Introduction to Philosophy of Science: Theory Building and Conception"),* third ed., pp. 171-191. Otava, 1980.

[57] P.W. Oman and C.R. Cook, "Typographic Style Is More than Cosmetic," *Comm. ACM,* vol. 33, no. 5, pp. 506-520, 1990.

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

[59] D.L. Parnas and D.M. Weiss, "Active Design Reviews: Principles and Practices," *Proc. Eighth Int'l Conf. Software Eng.,* pp. 132-136, 1985.

[60] A.A. Porter, L.G. Votta Jr., and V.R. Basili, "Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment," *IEEE Trans. Software Eng.,* vol. 21, no. 6, pp. 563-575, June 1995.

[61] B. Regnell, P. Runeson, and T.E. Thelin, "Are the Perspectives Really Different? Further Experimentation on Scenario-Based Reading of Requirements," *Empirical Software Eng.,* vol. 5, no. 4, pp. 331-356, Dec. 2000.
[62] D. Rombach, "Controlled Experiment on the Impact of Software Structure on Maintainability," *IEEE Trans. Software Eng.,* vol. 13, no. 3, pp. 344-354, Mar. 1987.
[63] P. Runeson and C. Wohlin, "An Experimental Evaluation of an Experience-Based Capture-Recapture Method in Software Code Inspections," *Empirical Software Eng.,* vol. 3, no. 4, pp. 381-406, 1998.
[64] P. Runeson, C. Andersson, T. Thelin, A. Andrews, and T. Berling, "What Do We Know about Defect Detection Methods?" *IEEE Software,* vol. 23, no. 3, pp. 82-90, May/June 2006.
[65] G.W. Russell, "Experience with Inspection in Ultralarge-Scale Development," *IEEE Software,* vol. 8, no. 1, pp. 25-31, 1991.
[66] C.B. Seaman, "Qualitative Methods in Empirical Studies of Software Engineering," *IEEE Trans. Software Eng.,* vol. 25, no. 4, pp. 557-572, July/Aug. 1999.
[67] H. Siy and L. Votta, "Does the Modern Code Inspection Have Value?" *Proc. Int'l Conf. Software Maintenance,* pp. 281-289, 2001.
[68] 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 and Reliability,* vol. 12, no. 3, pp. 155-171, 2002.
[69] T. Tenny, "Program Readability: Procedures versus Comments," *IEEE Trans. Software Eng.,* vol. 14, no. 9, pp. 1271-1279, Sept. 1988.
[70] K.E. Wiegers, *Peer Reviews in Software.* Addison-Wesley, 2002.
[71] M. Wood, M. Roper, A. Brooks, and J. Miller, "Comparing and Combining Software Defect Detection Techniques: A Replicated Empirical Study," *Proc. Sixth European Conf. Held Jointly with the Fifth ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* pp. 262-277, 1997.
[72] Z. Xing and E. Stroulia, "Refactoring Practice: How It Is and How It Should Be Supported—An Eclipse Case Study," *Proc. 22nd IEEE Int'l Conf. Software Maintenance,* pp. 458-468, 2006.
[73] Y. Zhou and H. Leung, "Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults," *IEEE Trans. Software Eng.,* vol. 32, no. 10, pp. 771-789, Oct. 2006.

Mika V. Mäntylä received the MSc degree in software engineering and the LicSc degree from Helsinki University of Technology, Espoo, Finland, in 2003 and 2005, respectively. He is currently finalizing his doctoral dissertation. He is a researcher and a project manager in the Software Business and Engineering Laboratory at Helsinki University of Technology. Previously, he was a software developer in the Finnish software industry. His research interests include empirical software engineering, software quality assurance, and software evolution.

Casper Lassenius received the DSc (Tech) degree in software engineering in 2006. He is a teaching researcher in the Software Business and Engineering Laboratory at Helsinki University of Technology and the head of the software process research group. His research interests include software measurement, software product development, agile development, and globally distributed software development. He is a member of the IEEE and the IEEE Computer Society.