Modern Code Reviews in Open-Source Projects: Which Problems Do They Fix?

Moritz Beller,
Alberto Bacchelli, Andy Zaidman
Delft University of Technology
The Netherlands
{m.m.beller, a.bacchelli, a.e.zaidman}@tudelft.nl

Elmar Juergens
CQSE GmbH
Germany
juergens@cqse.eu

ABSTRACT

Code review is the manual assessment of source code by humans, mainly intended to identify defects and quality problems. Modern Code Review (MCR), a lightweight variant of the code inspections investigated since the 1970s, prevails today both in industry and open-source software (OSS) systems. The objective of this paper is to increase our understanding of the practical benefits that the MCR process produces on reviewed source code. To that end, we empirically explore the problems *fixed* through MCR in OSS systems. We manually classified over 1,400 changes taking place in reviewed code from two OSS projects into a validated categorization scheme. Surprisingly, results show that the types of changes due to the MCR process in OSS are strikingly similar to those in the industry and academic systems from literature, featuring the similar 75:25 ratio of maintainability-related to functional problems. We also reveal that 7–35% of review comments are discarded and that 10–22% of the changes are not triggered by an explicit review comment. Patterns emerged in the review data; we investigated them revealing the technical factors that influence the number of changes due to the MCR process. We found that bug-fixing tasks lead to fewer changes and tasks with more altered files and a higher code churn have more changes. Contrary to intuition, the person of the reviewer had no impact on the number of changes.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Code inspections and walkthroughs

General Terms

Human Factors, Experimentation, Measurement, Verification

Keywords

Code Review, Open Source Software, Defects

MSR '14, May 31 – June 1, 2014, Hyderabad, India

1. INTRODUCTION

Code review is a widely agreed-on best practice in Software Engineering [11], consisting in the manual assessment of source code by human reviewers. It is mainly intended to identify defects and quality problems in the source code before its deployment in a live environment. We differentiate Formal Inspections from Modern Code Review (MCR, [5]): Inspections are heavyweight processes mandating a waterfall-like procedure, including an expert panel, a distinguished group meeting, and other formal requirements [19]; MCR is a lightweight process involving fewer formal requirements, a shorter time to a finished review, and often review tool support [22, 23, 29, 41]. Today, MCR dominates in practice [30], and is found both in industrial and OSS projects.

Due to the time-consuming nature of code reviews, researchers investigated their benefits and influencing factors [7, 36, 50]. While a variety of studies focused on inspections, research on MCR is in the early stages. Only two studies analysed the outcome of MCR: (1) Bacchelli and Bird investigated motivations and outcomes of MCR at Microsoft [5], and (2) Mäntylä and Lassenius classified the types of defects found in review on university and three industrial software systems [32].

Both studies investigate the defects that the MCR process *finds*, by analysing the comments written by reviewers. The analysis of the review comments, however, does not allow one to evaluate the problems that MCR *fixes*. In fact, comments might be disregarded, not clear in their target, or lead to unexpected changes; moreover, changes might not be triggered by comments, but by the authors themselves, or by an oral discussion. A precise evaluation of the problems that code reviews fix requires an in-depth investigation of the actual code changes that occur between the code submitted to review and the code eventually accepted. In this paper, we present such a study, investigating what kind of issues are fixed in MCR.

To test how previous findings from industrial and academic contexts apply, and to grant the replicability of our experiment, we focused on OSS systems for our investigation. We examined over 30 mature OSS systems claiming to do mandatory, continuous code review as possible study candidates. Unexpectedly, only two projects adhered to their claim. For these two systems, we manually classified more than 1,400 review changes into a defect categorisation, which we validated in an interrater-reliability study including two developers of those systems. We classified each change manually with the help of an Eclipse plug-in that we developed. Using the same setup, in a second step, we classified the trigger for the change.

Surprisingly, we found that the distributions of change types from our studies are strikingly similar to the defect distributions from the two prior studies: The reported 75:25 distribution of maintainability versus functional changes holds in our OSS systems, too. Similar to Mäntylä's results [32], the dominant change categories are code

comments (20%) and identifiers (10%). However, our study shows that 7–35% of review comments are discarded, and that a substantial 10–22% of the total changes are not triggered by review suggestions; this is not considered in prior studies.

While classifying changes, we noticed patterns in the meta-data (*e.g.*, bug-fixing tasks often had fewer changes than tasks implementing new functionality). We built a regression model on the technical influences of the review process to detect what could influence the number of changes made in reviews. We show that bug-fixing tasks lead indeed to fewer changes and that tasks with more altered files and a higher code churn have more changes on average. Interestingly, the reviewer has no impact on the number of changes. In interviews, developers confirmed our results match their intuition.

Structure of the paper: Section 2 introduces a common nomenclature on code reviews and provides an overview of the related work. Section 3 details our methodology: It introduces our initial research questions, the subject systems and the steps we took to conduct our investigation; subsequently, it details another research question, which emerged from our manual classification, and the steps to answer it. Section 4 presents the results for our research questions. We discuss the findings in Section 5 and conclude in Section 6.

2. BACKGROUND

We define a common nomenclature on abbreviations we use:

TMS (Task Management System): A software system to collect and administer coherent modifications in the form of task, such as Bugzilla[1] and Redmine.[2]

Task: The entity in which change requests are stored in a TMS. Synonyms are Issue or Ticket.

VCS (Version Control System): A software system where typically source code is stored in a repository with a retrievable history, *e.g.*, SVN[3] or Git.[4]

In the remainder of the section, we present the MCR process and give an overview over related work.

2.1 The Modern Code Review Process

From a black box perspective on Figure 1, a review is a process that takes as input *original source code* (*i.e.*, the first unreviewed change attempt, Step 1), and outputs *accepted source code* (2). The *author* is the person responsible for the implementation of the assigned task as source code. The *reviewer(s)* assure the implementation meets the quality standards. The original source code is a work that stemmed solely from the author, whereas in the accepted source code the author incorporated the reviewers' suggestions so that everybody is satisfied with the result.

The grey area in Figure 1 reveals the inner workings of the review process from a white box perspective: Once source code is submitted for review, the reviewers decide whether they accept it (3) or not (4). Their decision is normally based on the project's quality acceptance criteria, reviewing checklists, and guide lines. If they do not accept the code, reviewers annotate it with their suggestions (4) and send the *reviewed source code* back to the author. Addressing the reviewers' suggestions, the author makes alterations to the code and sends it back for further review (5). A *review round* comprises the two steps 'source code submitted for review' (1 or 5) and its actual, technical review (3 or 4). Therefore, a piece of code can minimally have one review round, if (3) is executed directly, or potentially infinitely many rounds, if the reviewers are never satisfied.

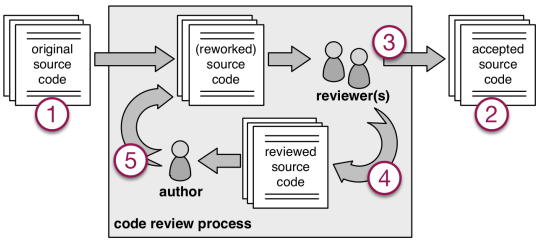

Figure 1: The review process detailed.

2.2 Related Work

Code reviews became subject to scientific examination with Fagan's 1976 paper on formal inspections [19]. Kollanus and Koskinen give an overview over the past and the status quo of research on reviews [30]. Overall, they note an evident lack of empirical knowledge on code reviews, and suggest more studies to be conducted to measure the effects of different review processes. With this paper, we strive to expand the current evidence-based knowledge on MCR.

There are two principal categories of code reviews: Formal, heavyweight review processes, called *inspections* (*e.g.*, the Fagan Inspection [19]), and lightweight code reviews with an emphasis on efficiency, referred to as *modern code reviews* [5, 30].

Formal Inspection. Despite their initial success, Fagan inspections have several disadvantages that hinder their continuous and widespread use across organizations: They mandate a number of formal requirements that do not adapt well to agile development methods, most notably a fixed, formal, waterfall process [33]. Several studies have shown that review meetings do not improve defect finding [10, 36, 45, 50]. Only one study reported contrary results [17].

Consequently, researchers and practitioners developed more lightweight, ad hoc code reviewing processes to better suit environments with test-driven and iterative development [6, 9, 34, 37, 47].

Modern Code Review. MCR is characterized by fewer formal requirements, a tendency to include tool support, and a strive to make reviews more efficient and less time-consuming. These features allowed many organizations to switch from an occasional to a mandatory, continuous employment of reviews [5]. MCRs often leave out the team meeting, and reduce the number of people involved in the review process to two. Adversely, Wood *et al.* found that the optimal number of reviewers should be two [54].

Review Effectiveness and Efficiency. Most research comparing inspection with testing and pair programming considers only functional defects [4, 28, 30, 44]. Given a reported 75% of non-functional defects in MCR [32], it stands to question whether the results from prior studies apply to modern review.

Sauer *et al.* [46] argue that individual expertise is the key factor in review effectiveness. Hatton [27] supports it: He found stark differences in defect finding capabilities among reviewers.

The ability to understand source code and perform reviews is called "software reading" [14]. The idea for this came from Porter and Votta [42], who advocated scenario-based reading, instead of generic checklists. Several code reading techniques such as defect-based reading and use-based reading have been suggested to educate code readers [14, 53]. El Emam and Wieczorek depict a code review process based on classic checklists [18].

[1] `http://www.bugzilla.org`
[2] `http://www.redmine.org`
[3] `http://subversion.apache.org`
[4] `http://git-scm.com`

Supporting Tools. MCR is often supported by tools, preferably integrated into the development environment (IDE) [12]. Bernhart *et al.* introduced one such tool for the Eclipse IDE, ReviewClipse [9], now Mylyn Reviews [39]. ReviewClipse automatically creates a new 'review process,' assigns a fitting reviewer, and opens a compare viewer for this commit. Reviewers perform reviews on the changed file in the IDE after it has been pushed into the VCS.

A popular review tool is the OSS Gerrit [22], offering web-based reviewing for projects using Git. It integrates the changes into the VCS only after the reviewer expressed consent to it [35].

A number of other review tools exist: *Mondrian*, a tool that Google uses for its closed-source projects [29]. *Phabricator* is Facebook's open-sourced tool [41]. *Github's review system* works with pull requests, comprising the code, a referenced task and possibly review comments [23]. Microsoft develops and uses *CodeFlow* [5].

3. METHODOLOGY

In this section we define the research questions, describe our case study objects, and outline our research method. More technical details can be found in the master's thesis origin of this work [8].

3.1 Research Questions

Our overall research goal is to gain an in-depth, analytical understanding of the outcome of the Modern Code Review process in OSS. This led to formulating the first two research questions. While working on them, we noted obtrusive patterns in the meta-data of our changes (*e.g.*, bug-fixing tasks had fewer changes than tasks implementing new functionality) that led to research question 3.

RQ1: Which changes occur in code under review?

RQ2: What triggered the changes occurring in code under review?

RQ3: What influences the number of changes in code under review?

3.2 Subject Systems

To answer our research questions empirically, we need suitable study objects. In this section, we motivate the choice of our two study objects and demonstrate their heterogeneity.

In total, as candidates for our case study, we examined over 30 OSS projects, which had some kind of claim to do code reviews. We found these projects by a Google search with the terms 'open source' and 'review', by harvesting Gerrit's show case projects,[5] and by posting in mailing lists and forums.

We discovered that many well-known OSS projects (*e.g.*, LibreOffice[6]) mostly review only code written by new developers. While they often seemingly employ code review tools, an examination of most projects' code review databases reveals that they are empty. For example, although Eclipse is listed as one of the show cases for Gerrit, none of the Eclipse projects enforce its usage. Tycho[7] is an Eclipse sub-project One of Tycho's main committers told us that "*if there is no functional change, there is no need to have an entry [in Gerrit]*" and "*we don't require a review – only that committers offer the chance for review. This is typically done by proposing a change in Gerrit and by waiting three days before submitting it.*" Consequently, a top Eclipse developer publicly admitted that he "*feels scared*" when committing without a review [49].

Since we want to examine the effects of continuously applied code reviews, we had to discard the majority of inspected study candidates. We finally chose ConQAT and GROMACS, which have a documented history of mandatory code reviews for every committed code change. Table 1 summarizes their main characteristics.

Table 1: Comparison of ConQAT and GROMACS.

	ConQAT[8]	GROMACS[9]
Access to Code	Public releases	Public VCS
Access to TMS	On request	On the website
Development time	≥ 8 years	≥ 18 years
# of Developers	12 active, ~50 total	~16 active, ~44 total
# of Reviewers	5 active	13 active
Language	Java	C (mostly)
SLOC	260,465	1,449,440
# of Tasks	~2,500	~1,200
Code Reviews since	2007	2011
Review mandatory	Yes	Yes
Review tool	None (Eclipse)	Gerrit
# of Reviewers/Task	1	≥ 2
# of Review Rounds	[1; ∞[	[1; ∞[
Samples	ConQAT-rand, ConQAT-100	GROMACS-rand

ConQAT. ConQAT is "*an integrated toolkit for creating quality dashboards that allow to continuously monitor quality characteristics of software systems.*"[10] Reviewers adopt a review-after-commit workflow: The reworked and reviewed source code versions are committed directly to the main development line in the VCS; reviewers use Eclipse to assess the source code and denote their suggestions as code comments. An important benefit of ConQAT is that we could establish a contact with its developers, thus we could approach the person responsible for any review change we were analysing

GROMACS. GROMACS is "*a versatile package to perform molecular dynamics,* i.e.*, simulate the Newtonian equations of motion for systems with hundreds to millions of particles.*"[11] Different from ConQAT, GROMACS uses Gerrit, so only the accepted source code is merged with the main development line. Reviewers enter their suggestions via a web-based user interface. We could not obtain direct access to the developer team.

3.3 Research Method – RQ1 and RQ2

Having selected the study subjects, we continued our investigation to answer RQ1 and RQ2. In the following, we detail the research method we adopted.

Change Classification. To understand which types of problems are fixed in the MCR process, we classify the changes that happen to code under review. We found that defect classifications in the literature are well-suited to characterize these code changes, although we use a broader definition ('change' instead of 'defect'). Due to the human-centric nature of code review, the term 'change' is more accurate than 'fix' or 'defect', as an improvement in the reviewed version might not be recognized as such by other reviewers. However, we used these terms because (1) they demonstrate the difference between review comments and review fixes clearly, (2) they are established and (3) they indeed fix problems according to the reviewer's perception. As a first step, we surveyed existing defect classifications. Researchers have produced an abundance of defect classifications [51], leading to an IEEE standard in 1993 [1]. The standard was the basis for two classifications by IBM and HP [13]. Wagner *et al.* [52] and Duraes and Madeira [16] evaluated these classifications and found them too general; consequently, El Emam and Wieczorek refined them [18]. Mäntylä and Lassenius based their

[5] `https://code.google.com/p/gerrit/wiki/ShowCases`
[6] `http://www.libreoffice.org`
[7] `http://projects.eclipse.org/projects/technology.tycho`
[10] `http://www.conqat.org`
[11] `http://www.gromacs.org`

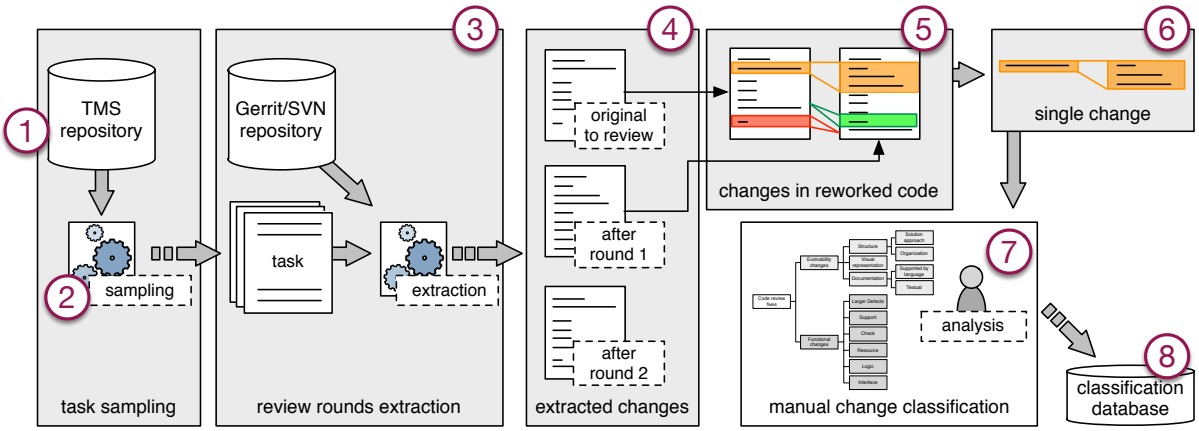

Figure 2: Research method applied to classify changes of code under review.

topology on this empirically validated classification scheme [32]. They analysed code review comments and extrapolated the types of defects found, reporting a 75:25 ratio between maintainability and functional defects. We found their classification to be applicable to the changes we observed – and not only to review comments. Our classification builds upon their work with small adaptations.

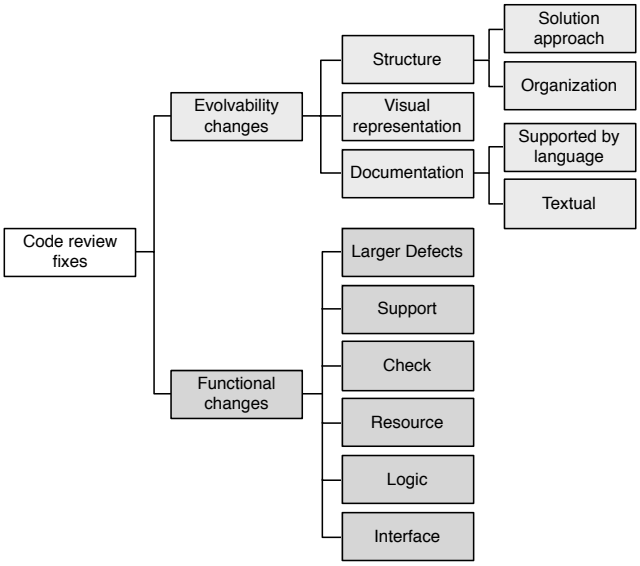

Figure 3: The taxonomy of changes of code under review.

According to our classification in Figure 3, a change can have implications in the form of a functional alteration in the software system, in which case it is a *functional* change. If it has none, it is a non-functional *evolvability* or *maintainability* change. We refine each of the two top-level categorizations into sub-groups.[12] *Structure* changes address problems that alter the compilation result of the code. They represent a category difficult to fix, as they require a deep understanding of the system under review. *Visual Representation* changes contain code-formattings without an effect on the compilation result. *Documentation* means documentary deficiencies (*e.g.*, comments and names) in the program text.

Our adaptations to the original classification are few: We included clarifications for Java-specific language constructs like 'assert' statements (often Debug Info) and the 'abstract' modifier (Visibility) so that others could understand these categories. We removed all subcategories in the resource category because these changes were rare. Most important, we removed the false positive category. Within the scope of our study, we find it an orthogonal concept to the type of a change: Per definition, either a change happened, and then we can categorize it in the appropriate category, or no code change happened, but then it is also not a false positive.

Motivation Classification. The motivation to do a change can be either (1) triggered by a *review comment* (as in Figure 4), or it can be (2) *undocumented.* If undocumented, the motivation can come from the author himself, the reviewers (*e.g.*, given orally), or from other sources.

Classification Process. Figure 2 depicts our classification approach, where we started at the level of tasks in the TMS (Step 1). By observing all code changes related to a task, we gain an understanding of the effects of code review on self-contained units of changes in the system. If we had just randomly selected unrelated code reviews from the code review population in the system, we would not have captured this relationship. Additionally, we could not have answered RQ3 with the traditional approach of entirely random review selection.

Task Sampling. ConQAT and GROMACS have too many tasks in the TMS to rate them all manually (see Table 1), thus we resorted to representative samples (Step 2). From ConQAT, we built two samples: A stratified random sample consisting of 120 tasks which aims to be a representative extract of all the ConQAT tasks, *ConQAT-rand.* Additionally, we inspected the 100 most-recently closed tasks to be able to unveil recent trends in code reviews as deviations from our representative sample ConQAT-rand, called *ConQAT-100.* Since GROMACS had fewer tasks with code reviews, we could extract only one sample: *GROMACS-rand.*[13]

Scope of Changes. Having selected the tasks to categorize, we performed a lookup to find the corresponding review rounds (Step 3) in either Gerrit (for GROMACS) or a pre-release of Teamscale.[14] Teamscale provided us with a consistent SVN history of ConQAT, in

[12] A more detailed description of the subgroups is given in `http://figshare.com/articles/Code_Review_Defects/689805`. El Emam *et al.* describe sub-categories of functional defects [18].

[13] A second sample (*i.e.*, GROMACS-100) would have 80% overlap with GROMACS-rand, rendering it of little additional value.

[14] `http://www.teamscale.org`

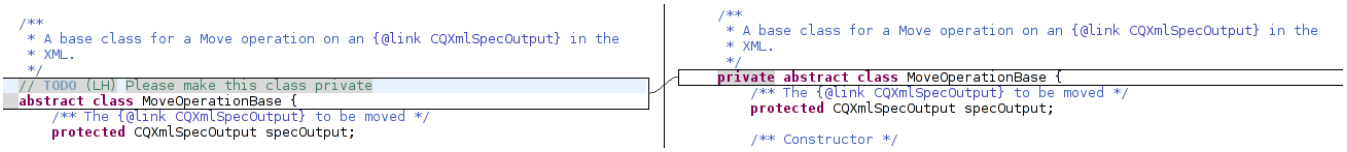

Figure 4: A review-triggered code change.

which we could query all commits made to a specific task. Similar to Gerrit, this allowed us to infer all review rounds for each task individually (Step 4). Once we identified the review rounds, we could compare the reworked source code to the reviewed source code of the prior round (Step 5). In practice, we did a file-by-file comparison of all files touched in the review process. We used our own Eclipse plugin and Gerrit's web-compare view for this. With these tools, we identified the scope of all modifications, *i.e.*, which textual differences comprised one logical, cohesive change in the sense of our categorisation from Figure 3 (Step 6). This was trivial as long as the change was triggered by a review comment. However, for undocumented changes, it was sometimes harder to decide the scope, and thus the adequate change category. In these cases we asked ConQAT's developers to help us in defining the scope and categorisation (Steps 6 and 7). In the few cases where developers could not make sense of the modifications, or the original developer was not available, we invalidated the task. For GROMACS, we immediately invalidated the task.

Change Classification. Once we had separated all changes in Step 6, we performed two independent ratings for each change in Step 7.

For RQ1, we rated the type of one code change in exactly one category, supported by our change classification definition. If we considered more than one defect category suitable, we used the most precise fitting, *i.e.*, the one which explained best why a change was conducted. While we tried to rate modifications as fine-granular and precise as possible, we preferred to rate bigger changes with a recognizable functional modification as one larger functional change. If we could rate a change as either evolvable or functional, we preferred the latter, in accordance with Mäntylä and Lassenius [32]: We consider functional implications more severe, and most functional changes inevitably have maintainability implications.

For RQ2, we assessed the motivation to do the change, either as a review comment or as undocumented. Furthermore, we denoted in Step 6 if a review comment did not lead to a change because the author and reviewer decided to abandon it.

Databases. In the last step (Step 8), we stored each categorization together and the meta-data about review round and task, in a database. In an effort to stimulate replication and further research on MCR, we make these databases publicly available.

3.4 Subject System for RQ3

While classifying review changes for RQ1 and RQ2, we noted some patterns in ConQAT's data. For example, tasks categorised in the TMS as bug fix had fewer changes under review than tasks categorised as new feature implementation. This left us wondering which factors influence the number of changes in ConQAT (RQ3).

In our answer to RQ3 we did not want to be limited to a purely quantitative study, but wanted to qualitatively interpret the observed coefficients. Therefore, we selected only ConQAT as study subject. Interviews with developers and architects would help us link quantitative and qualitative information to gain a deeper understanding of the influences leading to more or fewer changes.

3.5 Research Method – RQ3

Intuitively, we assume that the amount of changes under review depends on the person of the reviewer and author, the type of task, how extensive the alterations to the systems were, and other influences. To capture possibly all relationships, we performed interviews with ConQAT developers and reviewed the influences proposed in literature (*e.g.*, [7]). These are the explanatory variables in the model, and their possible influence on the number of changes:

Code Churn (discrete count, d.s.) $\in [0; \infty[$
Code churn is a metric of how many textual changes occurred between two versions [38]. The larger the code churn in the original file, the more there is to review and therefore, the more changes could follow.

Number of Changed Files (d.s.) $\in [0; \infty[$
The more wide-spread a change is across files, the more concepts it touches in a system. It is difficult to master all these concepts, and this could make more rework necessary.

Author (categorial, c.) $\in$ {bader, beller, besenreu, deissenb, ...}
The author of the original code. We suppose certain authors to need more changes during review than others.

Task Type (c.) $\in$ {uncategorized, adaptive, corrective, ...}
The task type describes the kind of work that is expected to occur in a task [26]. Corrective tasks are bug fixes. Perfective are tasks that implement new functionality. Preventive tasks shall simplify future modifications; adaptive tasks adapt the system to changes in the execution environment; ConQAT developers had a common understanding of corrective and perfective tasks, but preventive and adaptive were less clear, and also less used [43]. Uncategorised is the default, and also used for tasks that do not fit in any of the other categories. Developers set the type for each task manually in the TMS.

Package (c.) $\in$ {edu.tum.cs.conqat.ada, ...}
ConQAT is internally structured into more than 30 different top-level packages. Review on parts of the ConQAT engine is believed to be rigorous, while review in the IDE parts might be laxer. This variable reports the main building site of a task.

Reviewer (c.) $\in$ {heinemann, hummel, juergens, ...}
We presume that the reviewer has one of the largest influences, since the review suggestions are his work. We could imagine some reviewers to be more strict than others.

Dependent Variable. The dependent variable for our model is the number of changes due to code review. It is far more fine-grained than the number of review rounds.

Data Sampling. Based on the above description of the characteristic features of these variables, we developed algorithms to automatically gather the data. In contrast to RQ1 and RQ2, we could analyse the complete ITS this way. In total, we analysed 2,880 changes under review in 973 tasks from ConQAT.

Application of Regression Analysis. To analyse the impact of the aforementioned influences, we use a generalised linear model (GLM). Mixed models or GLMs are a common methodology in Software Engineering research [4]. Such a regression model applies because it describes characteristics of a dependent variable – number of changes – in terms of explanatory variables $X_1 ... X_n$ in retrospect, which is the case for our data. In contrast to linear models, GLMs can handle both cardinal and discrete count variables, which makes them ideal for the analysis of our model.

We evaluated the GLM with the help of the statistics software R. For the evaluation, we first determined a distribution that best fits our dependant variable. The histogram for the distribution of the changes is similar to a Poisson distribution, but is zero-inflated and skewed to the left. A GLM is the preferred approach for such non-normal distributions of the dependent variable [55]. Following this advice, we modelled the number of changes with a negative binomial distribution [31]. The theory of GLM dictates that no strong or trivial relationship in-between the explanatory variables should exist. We calculated Pearson's r as an estimator of the relationship between explanatory variables that are likely correlated and found a weak correlation between code churn and number of changed files ($r = 0.33$ at a 0.95 confidence interval), which does not impede the applicability of GLM [20].

4. RESULTS

In this section, we report the results of our case studies to RQ1–3.

4.1 Types of Changes in Reviewed Code

In RQ1, we researched which types of problems are fixed in MCR. We give an overview of the findings, and then focus on the results for maintainability versus functional changes, and the detailed results from manual classification.

Table 2: Task and review change distribution in RQ1 and RQ2.

Metric	ConQAT-rand	ConQAT-100	GROMACS
# of valid Tasks	100	89	60
# of Changes	892	361	216
Average	8.81	4.00	3.24
Median	2	0	0
Max. #Changes/Task	208	110	93

Number of Changes. Even after normalising the number of tasks to 100 (see Table 2), we note a discrepancy in the number of changes in our three samples: There were relatively fewer changes during review in ConQAT-100 and GROMACS than in ConQAT-rand. We would have expected similar results from ConQAT-rand and ConQAT-100, but we found a stark difference: 892 changes in the former and 361 in the latter. In GROMACS we found only 216 changes. A related characteristic is the number of changes *per task*. As Table 2 shows, the distribution is skewed towards a small number of changes, with few extreme outliers.

Evolvability vs. Functional Changes. Figure 5 presents the ratio of evolvability and functional changes in our three data sets, and puts it in context with the values reported by Mäntylä and Lassenius [32]. The ratios among the four systems are similar, floating around 75:25, within a range of ten percentage points.

In ConQAT-rand we found more evolvability fixes than in all other samples, 6% points above 75%. ConQAT-100 hits the 75:25 ratio almost exactly. GROMACS has a slightly lower amount of evolvability changes at 69%. The uniformity of the result is surprising, because ConQAT and GROMACS are written in different programming languages and development models by different people with diverging review processes.

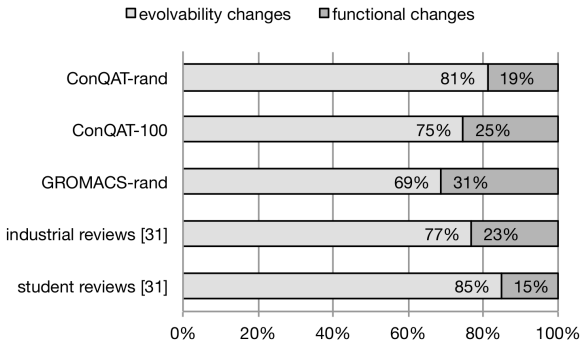

Figure 5: Evolvability vs. functional changes (RQ1).

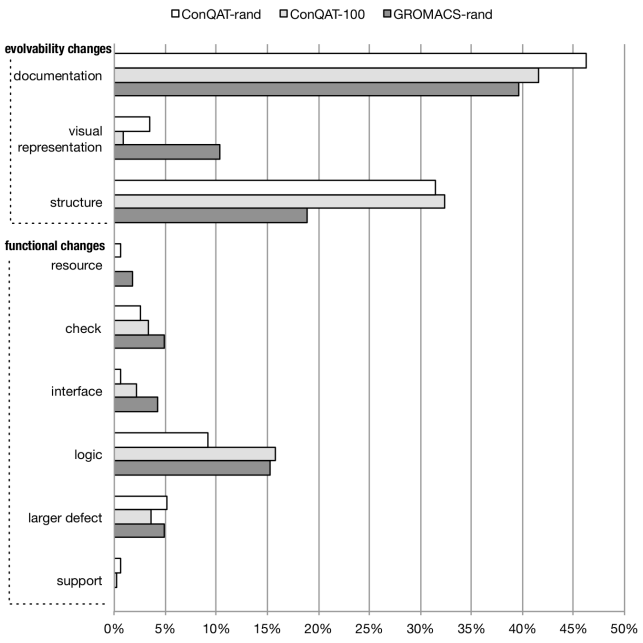

Figure 6: Change distribution profiles (RQ1).

Detailed Change Profiles. Figure 6 depicts the detailed study results: It lists the frequencies of each change category from Section 3.3.[15]

Following the trend from the top-level groups, the normalised distribution profiles of ConQAT-rand, ConQAT-100, and GROMACS-rand, look comparable. For example, we find changes in code comments to be the single highest change category across all the profiles, followed by identifier renamings in ConQAT, and 'other' functional changes, and then renamings in GROMACS (categories 'documentation' and 'logic'). In GROMACS no changes from the 'documentation-language' subcategory were fixed, because GROMACS is a C system, and C does not support the object-orientated concepts specific to this subgroup. In ConQAT, few changes stem from the 'visual representation' subgroup, as developers use Eclipse's auto-formatter to fix these problems.

[15]The fine-grained, absolute distribution profiles are available on `http://dx.doi.org/10.6084/m9.figshare.915389`.

4.2 Triggers of Code Changes

RQ2 regards what triggers a change in reviewed code. Additionally, this includes how many review comments are realised, and how many are discarded.

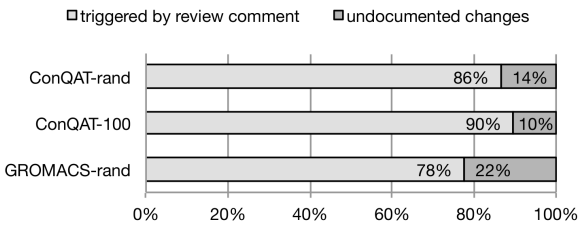

Figure 7: Distribution of the triggers for code changes (RQ2).

Figure 7 illustrates the results. ConQAT-rand, ConQAT-100, and GROMACS show two uniting features: Most changes under review come from review suggestions (78–90%). Yet, undocumented changes, which have not been captured in prior research, represent more than 10% of all changes (10–22%). Interviewed ConQAT developers explained that undocumented changes are almost exclusively self-motivated: During a review the author autonomously decides to do a change that was not requested by any reviewer.

The acceptance of review comments is diverging: In the ConQAT samples, more than 90% of review suggestions lead to a change (93% for ConQAT-rand and 91% for ConQAT-100). In GROMACS, only 65% of review suggestions are realised, while the remainder is discarded in unison by the author and the reviewer.

We noticed a relationship between the presence of review-triggered and undocumented changes: Whenever there were undocumented changes in a task, at least one review-triggered change happened, too.

We explored a relationship that links the type of changes and their motivation, combining RQ1 and RQ2. In both ConQAT samples a strong correlation is present between review-triggered changes and comment type changes ($\rho_{ConQAT-rand} = 0.74$, $\rho_{ConQAT-100} = 0.71$), which is weaker for undocumented changes ($\rho_{ConQAT-rand} = 0.43$, $\rho_{ConQAT-100} = 0.35$ at a 0.95 confidence interval). The correlation is moderate in GROMACS, as no such evident difference exists between review-triggered and undocumented comment changes ($\rho_{GROMACS} = 0.56$ for the first, $\rho_{GROMACS} = 0.51$ for the latter).

4.3 Influences on the Number of Changes

The regression analysis yields a set of coefficient values for each of the explanatory variables in our model. We report on how we evaluated the model and the results with the 973 sampled tasks from ConQAT.

First Model Fit. Our first model fit reported a $\theta = 0.4124$, a standard error of 0.0328 changes and a log-likelihood of –1,487.5. The benefit of the standard error is that – in contrast to R^2 – it tells us how strong the observed values differ on average from the regression model *in the unit of the dependent variable* [20]. In comparison to the unit-less R^2, this provides us with an intuitive understanding of the model accuracy: On average, the calculated number of changes is off by 0.03.

The detailed coefficient results showed that we did not have one reviewer or author instantiation for the variable 'reviewer' and 'author' with a statistically significant value [2]. Therefore, we question whether the 'reviewer' or 'author' parameter as a whole have an overall significant impact on the dependent variable 'number of changes'.

To test for this, we performed a 14 degrees of freedom χ^2-test [24], in which we compared the model with and without the explanatory variable 'reviewer'. At $Pr(\chi) = 0.23$, the χ^2-test implied that the 'reviewer' is a statistically insignificant predictor of the number of changes, because it is larger than the significance level of 0.05. As a result, we refined our model to exclude the 'reviewer'. The variables 'author' and 'package' have a significant influence, confirmed by the according χ^2-tests.

Second Model Fit. Our second model fit reported $\theta = 0.4001$, a standard error of 0.0317 changes and a log-likelihood of –1,496,512. Equation 1 expresses the refined influence model (without 'reviewer'), where i is the task number.

$$\begin{aligned}\log(\text{Number of Changes}_i) = \epsilon + \beta_1 \cdot \text{Code Churn}_i + \\ + \beta_2 \cdot \text{Number of Changed Files}_i + \\ + \beta_3 \cdot \text{Package}_i + \beta_4 \cdot \text{Task Type}_i + \beta_5 \cdot \text{Author}_i \end{aligned} \quad (1)$$

We report ϵ and the β coefficients along with their significance in Table 3. As we have an underlying logarithmic relationship, the effects of small parameter alterations (*e.g.*, number of changed files) can have greater effects than a simple linear function, part of the reason why some coefficients are relatively small (*e.g.*, β_1).

Table 3: Calculated model coefficients for RQ3 (see [8]).

Coefficient	Value	Significance Level
Error Term ϵ (Intercept)	-34.0042	1.0000
Code Churn β_1	0.0026	$2.34 \cdot 10^{-16}$ **
Changed Files β_2	0.0483	$< 2 \cdot 10^{-16}$ **
Package	β_3	
org.conqat.engine.blocklib	-36.9484	1.0000
org.conqat.engine.commons	0.7152	0.4649
org.conqat.ide.editor	0.1793	0.8633
org.conqat.ide.index.dev	-39.3108	1.0000
...		
Task Type	β_4	
adaptive	0.5277	0.1283
corrective	-0.6508	0.0496 *
perfective	0.7015	0.0138 *
preventive	-0.7289	0.1118
Author	β_5	
deissenb	32.4496	1.0000
juergens	32.7963	1.0000
...		

Significance codes: ** = 0.001, * = 0.05

5. DISCUSSION

In this section, we discuss our results and show how we mitigated the threats that endanger them.

5.1 Types of Code Changes

In RQ1, we asked: Which types of changes occur in code review? Our results reveal that most changes are related to the evolvability of the system (75%), and only 25% to its functionality (see Figure 3).

Change Types. Prior research reported similar ratios for industrial and academic systems [5, 32]. Therefore, the MCR process seems to be generally more valuable for improving maintainability, than for fixing defects. Development teams should be well-informed about this particular outcome, to better decide how to include MCR in their process. For example, MCR might be recommended for software systems that require high maintainability, such as long-lived software systems.

Code Review vs. Testing. Prior research compared the benefits of code review to testing by considering only their respective effectiveness in discovering functional defects [28]. Our study reveal that the other types of problems that are fixed with code reviews dominate over functional fixes (75% vs. 25%), thus an approach that considers only the functional fixes might lead to drawing wrong conclusions about the usefulness of the MCR process in general. Our results are backed-up by similar change type distributions reported by Mäntylä and Lassenius for industrial and academic reviews [32] (see Figure 1), further questioning the value of the partial comparison between code review and testing.

Number of Changes. Apart from the types of changes, we also observed some trends in the number of changes: Considering in order ConQAT-rand, ConQAT-100, and GROMACS, the normalised number of changes decreases. Developers' explanation for ConQAT-100 is that in this observation period many tasks stemmed from a well-rehearsed team of two developers, leading to fewer changes in reviews. For GROMACS, we attribute the fewer changes in review to not as-strict and detail-oriented reviews in comparison with ConQAT. Interestingly, this did not affect the distribution of change types in Figure 6.

Splitting Large Tasks. We observed that in GROMACS-rand and ConQAT-100 more than 50% of tasks pass review directly (median of 0, Table 1). Few tasks have an extraordinary amount of changes. As the variance is great it is hard for reviewers to estimate in advance how long and how difficult a review will be. We would expect that this problem could be solved by generating a more uniform distribution of review comments with the splitting of larger tasks into smaller sub-tasks. However, by analysing the tasks with many review changes (> 30), we found that the exact opposite happens in practice. Large tasks with many changes included the review of additional satellite tasks. In other words, related tasks started out small and separated (as we would have suggested initially), but were then merged during development or during review. From an economical point of view, it makes sense for one reviewer to also review related tasks. Still, reviewers would probably benefit from separated, smaller, and more manageable review chunks. Further analysis on this revealed that a technical reason is also preventing this chunking: The review process in ConQAT is file-based and related tasks often perform cross-cutting changes in the same files. At least parts of such tasks must be reviewed together (in ConQAT, this is determined via communication in the TMS), thus increasing the number of changes in one of the tasks. This is less frequent in GROMACS, where changes can be reviewed independently. This underlines the advantage of specialised tool support for the MCR process and the advantage of change-based over file-based reviews.

5.2 Triggers for a Code Change

In RQ2, we asked: What triggered the changes that occurred in code under review? We showed that in 78–90% the trigger are review comments. The remaining 10–22% are 'undocumented.'

Self-Motivated Changes. In developer interviews for ConQAT, we revealed that almost all undocumented changes are in fact *self-motivated changes, i.e.*, they are made by the author without an explicit request from the reviewer. This is because ConQAT's review process demands that every oral review suggestion be at least also textually documented. As the GROMACS team is physically more spread-out, such out-of-the-band communication through mailing lists, group chats, fora or private communication is a concern we could not fully exclude [25]. However, we did not notice a lack of documentation for actions taken in Gerrit.

Triggering Self-Motivated Changes. We discovered that in each case where a self-motivated change is present, there needs to be at least one review-triggered change. In other words, we found that self-motivated changes are only made when the reviewer does not directly accept the modifications in the task, but includes some comments. We did not investigate this finding further, but studies can be designed and carried out to determine if and how other reviewers' comments give rise to autonomous reflection on the code under review and improved effectiveness of the MCR process.

Changes in Comments. While correlation is not causation, we assume that reviewers notice outdated and ill-fitting comments more readily than developers themselves. This could be an explanation for the increased correlation for comment changes in review-triggered changes vs. comment changes in self-motivated changes.

5.3 Influences on the Number of Changes

In RQ3, we asked: What influences the number of changes in code under review? We found that the reviewer did not have an influence on the number of changes, and that code churn, number of changed files and task type are the most important factors influencing the number of changes. In this section, we underline these numerical findings from Table 3 with qualitative explanations from developer interviews.

Reviewer Insignificant. Confronted with our observations, two ConQAT developers answered that it fits their experience and intuition that they do the same amount of rework for every reviewer. One developer answered that "*the amount is about the same, with some giving slightly more* [review comments] *than others.*" One developer answered that he does not have a gut feeling for this relationship. In his opinion, the lack of a feeling for this could be because he does not think about the reviewer when incorporating changes. All ConQAT developers speculate that the reason behind this finding is the long term collaboration between the developers – some developers have been working together for more than 8 years. One developer said that he was "*convinced that our review process creates a very common understanding of the way we code.*"

Size and Task Type Significant. We asked developers about the observations that (1) bug-fixing tasks lead to fewer changes than tasks which implement new functionality, and (2) tasks with more altered files and a higher code churn have more changes. One developer answered that he thinks "*the number of review comments correlates with the amount of* [altered] *code. This explains both observations.*" The answer implies that bug fixes affect fewer lines of code than tasks that implement new functionality. Osman *et al.* have shown that, indeed, most bug fixes concern only one line of code [40]. It is intuitive that a single line of code change cannot cause as many review objections as the development of a full-blown new feature.

Package. While the package where a review took place played a role, we have no significant data to support the assumption that there are generally more review changes in critical parts of ConQAT.

Author. A similar explanation as for the package is true for the author. The main reason for the lack of confidence in the calculated coefficients is the high variability in the amount of changes per author and task. Although the values are not significant, all main ConQAT developers have a similar coefficient. This would support the gut feeling of developers that all authors have the same amount of rework during review.

5.4 Threats to Validity

Both internal and external threats endanger the validity of our results. In this section, we show how we mitigated them.

Internal Threats. Internal threats concern the validity of our measurements. As such, the Hawthorne Effect refers to the phenomenon that participants of case studies perform above average because of

the knowledge that they are observed [3]. We could rule out this effect since we started our studies a-posteriori: Neither the authors nor reviewers from ConQAT or GROMACS knew we would undertake this study.

Due to stratified randomized sampling, we captured a representative sample of ten tasks per regular author. This way, no single author has an over-proportional impact on the result, and thus we avoid biased sampling. This way, we also exclude inexperienced authors from the study.

External Threats. External threats concern the generalisability of our results. While we assume ConQAT and GROMACS prototypical of current OSS projects that employ continuous reviews, the analysis of only two systems does not allow us to draw conclusions about OSS in general. We need a larger case study on more projects for this. However, with over 1,400 categorized review changes our study is, to the best of our knowledge, the largest manual assessment on reviews thus far.

The categorization process for RQ1 and RQ2 is subjective. The results are only generalisable if interrater reliability is given. We addressed this problem with a study that measured agreement between the four study participants and our own reference estimation with the κ measure [15] on 100 randomly-selected changes (at least 2 changes per change category). While we are aware of the limitations of the κ-measure and its interpretation [48], it is the default procedure for measuring interrater reliability. For RQ 1, we received κ values from 0.5 to 0.6 (*i.e.*, a "fair to good agreement" [21]) when considering the detailed change classification, and values from 0.5 to 0.8 when only differentiating evolvability and functional changes (*i.e.*, a "fair to excellent agreement"). For RQ 2, we received κs between 0.8 and 1.0 (complete agreement), excluding the threat that our classification is too subjective.

In RQ3, algorithms performed all measurements. Therefore, there might be the danger of a systematic bias. We used the manually extracted ConQAT data from RQ1 and RQ2 – a subset of the data for RQ3 – as a gold-standard to determine the quality. The comparison showed that the algorithms were accurate for categorial variables; they slightly underestimated numerical values, rendering our model parameters in Table 3 a safe lower boundary.

6. CONCLUSION

In this paper, we have shown that prior research on code reviews did not consider the entire outcome of MCR on reviewed code. Prior research has failed to capture up to 23% of the changes applied after review. Moreover, our investigation leads to the insight that 10–35% of review suggestions have not lead to changes in the code.

Despite this bias in previous studies, our change type distributions for two OSS systems show a strong similarity to the defect type distributions of prior industrial and academic reviews. We confirm a 75:25 ratio between evolvability and functional changes, which strengthens the case for code review in long-lived software systems that require high maintainability.

Moreover, we have found that the number of changes in reviews is independent of the reviewer in ConQAT. This unintuitive finding indicates that reviews, though people-driven, could be more people-independent than originally assumed. Instead, factors influencing how many changes need to be made in review comprise the type and the volume of the task: Bug-fixing tasks tend to have fewer changes than tasks that implement new functionality. The more code churn or the higher the number of touched files in a task, the more changes do we expect during review on average.

We have discovered a central shortcoming in the analysis of the actual benefits of modern code review and propose a methodology to overcome it. In addition, we have built and evaluated a model of the influences on the code review process. Our vision is that one day, code reviewers can input the amount of time they want to spend on review, and a tool will suggest the best-fitting task for them to review.

Contributions. In this paper, we made the following contributions:
Change Classification. A validated change classification for current Java and C projects to better understand the actual effects of code reviews on the software.
Case Study on Two OSS. A case study with two OSS that manually analyses the type of over 1,400 changes done in reviews: 75% of changes in reviews are maintainability related, only 25% concern functionality.
Motivation for Review Changes. A case study researching the motivation for the changes in review: The majority of changes in reviews is driven by reviewers' comments. But there is also a substantial amount of undocumented changes (10–18%), which prior research has neglected. Moreover, up to 35% of review comments are discarded.
Model on Code Review Influences. A model on code review influences: Reviewing homogeneously leads to the same number of changes independent of the reviewer, whereas the type of the task, the code churn and the number of touched files have a significant influence on how many changes are performed in a code review.

7. ACKNOWLEDGEMENT

We gratefully acknowledge the financial support of the The Netherlands Organisation for Scientific Research (NWO) for the project *TestRoots*, which partially sponsored this work.

8. REFERENCES

[1] IEEE standard classification for software anomalies. *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, pages 1–23, 2010.

[2] B. Abraham and J. Ledolter. *Introduction to regression modeling*. Thomson Brooks/Cole, 2006.

[3] J. Adair. The hawthorne effect: A reconsideration of the methodological artifact. *Journal of applied psychology*, 69(2):334, 1984.

[4] E. Arisholm, H. Gallis, T. Dyba, and D. Sjoberg. Evaluating pair programming with respect to system complexity and programmer expertise. *IEEE Transactions on Software Engineering*, 33(2):65–86, 2007.

[5] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 712–721. IEEE Press, 2013.

[6] R. Baker. Code reviews enhance software quality. In *Proceedings of the International Conf. on Software engineering (ICSE)*, pages 570–571. ACM, 1997.

[7] O. Baysal, O. Kononenko, R. Holmes, and M. Godfrey. The influence of non-technical factors on code review. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 122–131. IEEE, 2013.

[8] M. Beller. Quantifying Continuous Code Reviews. Master's thesis, Technische Universität München, 2013. `http://dx.doi.org/10.6084/m9.figshare.918618`.

[9] M. Bernhart, A. Mauczka, and T. Grechenig. Adopting code reviews for agile software development. In *Agile Conference (AGILE), 2010*, pages 44–47. IEEE, 2010.

[10] A. Bianchi, F. Lanubile, and G. Visaggio. A controlled experiment to assess the effectiveness of inspection meetings. In *Proceedings International Software Metrics Symposium (METRICS)*, pages 42–50. IEEE, 2001.

[11] B. Boehm and V. Basili. Software defect reduction top 10 list. *Foundations of empirical software engineering: the legacy of Victor R. Basili*, page 426, 2005.

[12] L. Cheng, C. de Souza, S. Hupfer, J. Patterson, and S. Ross. Building collaboration into ides. *Queue*, 1(9):40, 2003.

[13] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, and M. Wong. Orthogonal Defect Classification - A Concept for In-Process Measurements. *IEEE Transactions Software Engineering*, 18(11):943–956, 1992.

[14] M. Ciolkowski, O. Laitenberger, D. Rombach, F. Shull, and D. Perry. Software inspections, reviews and walkthroughs. In *Proceedings International Conf. on Software Engineering (ICSE)*, pages 641–642. IEEE, 2002.

[15] J. Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

[16] J. Duraes and H. Madeira. Definition of software fault emulation operators: A field data study. In *Prod. International Conf. on Dependable Systems and Networks (DSN)*, pages 105–114. IEEE, 2003.

[17] C. Ebert, C. Parro, R. Suttels, and H. Kolarczyk. Improving validation activities in a global software development. In *Proceedings of the International Conf. on Software Engineering (ICSE)*, pages 545–554. IEEE CS, 2001.

[18] K. El Emam and I. Wieczorek. The repeatability of code defect classifications. In *Proceedings International Symp. Software Reliability Engineering (ISSRE)*, pages 322–333. IEEE, 1998.

[19] M. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[20] L. Fahrmeir, T. Kneib, S. Lang, and B. Marx. *Regression: Models, Methods and Applications*. Springer, 2013.

[21] J. Fleiss. Statistical methods for rates and proportions, 1981.

[22] Gerrit. `https://code.google.com/p/gerrit/`. Accessed 2014/01/20.

[23] GitHub. `https://github.com/`. Accessed 2014/01/14.

[24] P. Greenwood and M. Nikulin. *A guide to chi-squared testing*, volume 280. Wiley-Interscience, 1996.

[25] A. Guzzi, A. Bacchelli, M. Lanza, M. Pinzger, and A. v. Deursen. Communication in open source software development mailing lists. In *Proceedings of the Tenth International Workshop on Mining Software Repositories*, pages 277–286. IEEE Press, 2013.

[26] J. Hartmann and D. Robson. Techniques for selective revalidation. *IEEE Software*, 7(1):31–36, 1990.

[27] L. Hatton. Testing the value of checklists in code inspections. *Software, IEEE*, 25(4):82–88, 2008.

[28] C. Kemerer and M. Paulk. The Impact of Design and Code Reviews on Software Quality: An Empirical Study Based on PSP Data. *IEEE Transactions Software Engineering*, 35(4):534–550, 2009.

[29] N. Kennedy. Google Mondrian: web-based code review and storage. `http://www.niallkennedy.com/blog/2006/11/google-mondrian.html`, 2006. Accessed 2013/12/16.

[30] S. Kollanus and J. Koskinen. Survey of software inspection research. *The Open Software Engineering Journal*, 3(1):15–34, 2009.

[31] C. Krebs. *Ecological methodology*, volume 620. Benjamin/Cummings Menlo Park, California, 1999.

[32] M. Mäntylä and C. Lassenius. What Types of Defects Are Really Discovered in Code Reviews? *IEEE Transactions Software Engineering*, 35(3):430–448, 2009.

[33] R. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[34] B. Meyer. Design and code reviews in the age of the internet. *Communications of the ACM*, 51(9):66–71, 2008.

[35] L. Milanesio. *Learning Gerrit Code Review*. Packt Publishing Ltd, 2013.

[36] J. Miller, M. Wood, and M. Roper. Further experiences with scenarios and checklists. *Empirical Software Engineering*, 3(1):37–64, 1998.

[37] H. Mills, M. Dyer, and R. Linger. Cleanroom software engineering. 1987.

[38] J. Munson and S. Elbaum. Code churn: A measure for estimating the impact of code change. In *Software Maintenance, 1998. Proceedings. International Conference on*, pages 24–31. IEEE, 1998.

[39] Mylyn Reviews. `http://www.eclipse.org/reviews/`. Accessed 2013/12/16.

[40] H. Osman, M. Lungu, and O. Nierstrasz. Mining frequent bug-fix code changes. In *IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 343–347. IEEE CS, 2014.

[41] Phabricator. `http://phabricator.org/`. Accessed 2014/04.

[42] A. Porter and L. Votta. An experiment to assess different defect detection methods for software requirements inspections. In *Proceedings International Conf. on Software Engineering (ICSE)*, pages 103–112. IEEE CS, 1994.

[43] T. Ritzau and J. Andersson. Dynamic deployment of java applications. In *Java for Embedded Systems Workshop*, volume 1, page 21, 2000.

[44] P. Runeson, C. Andersson, T. Thelin, A. Andrews, and T. Berling. What do we know about defect detection methods? *IEEE Software*, 23(3):82–90, 2006.

[45] G. Sabaliauskaite, S. Kusumoto, and K. Inoue. Assessing defect detection performance of interacting teams in object-oriented design inspection. *Information and Software Technology*, 46(13):875–886, 2004.

[46] C. Sauer, R. Jeffery, L. Land, and P. Yetton. The effectiveness of software development technical reviews: A behaviorally motivated program of research. *Software Engineering, IEEE Transactions on*, 26(1):1–14, 2000.

[47] H. Uwano, M. Nakamura, A. Monden, and K. Matsumoto. Analyzing individual performance of source code review using reviewers' eye movement. In *Proceedings of the 2006 symposium on Eye tracking research & applications*, pages 133–140. ACM, 2006.

[48] A. Viera and J. Garrett. Understanding interobserver agreement: the kappa statistic. *Fam Med*, 37(5):360–363, 2005.

[49] L. Vogel. Whenever i apply a change to the eclipse platform project without additional review, i feel a bit scared. `https://twitter.com/vogella/status/423799752833003521`, 2014.

[50] L. Votta. Does every inspection need a meeting? In *ACM SIGSOFT Software Engineering Notes*, volume 18, pages 107–114. ACM, 1993.

[51] S. Wagner. Defect classification and defect types revisited. In *Proceedings of the 2008 workshop on Defects in large software systems*, pages 39–40. ACM, 2008.

[52] S. Wagner, J. Jürjens, C. Koller, and P. Trischberger. Comparing bug finding tools with reviews and tests. In *Testing of Communicating Systems*, pages 40–55. Springer, 2005.

[53] Y. Wang, L. Yijun, M. Collins, and P. Liu. Process improvement of peer code review and behavior analysis of its participants. In *ACM SIGCSE Bulletin*, volume 40, pages 107–111. ACM, 2008.

[54] M. Wood, M. Roper, A. Brooks, and J. Miller. Comparing and Combining Software Defect Detection Techniques: A Replicated Empirical Study. In *Symposium on the Foundations of Software Engineering (FSE)*, pages 262–277, 1997.

[55] C. Wu and M. Hamada. *Experiments: planning, analysis, and optimization*, volume 552. John Wiley & Sons, 2011.