FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Computer Science

Quantifying Continuous Code Reviews

Moritz Marc Beller

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Computer Science

Quantifying Continuous Code Reviews

Quantifizierung kontinuierlicher Code-Inspektionen

Author:	Moritz Marc Beller
Supervisor:	Prof. Dr. Dr. h.c. Manfred Broy
Advisor:	Dr. Elmar Jürgens
Date:	October 15, 2013

I assure the single handed composition of this Master's Thesis, only supported by declared resources.

München, October 15, 2013 Moritz Marc Beller

Acknowledgements

This thesis is a mere dwarf on the shoulders of giants. It would not have been possible without the achievements of many a scientist, some of which did not receive their rightful attribution in their lifetime. I do want to express my sincere gratitude to the persons who are my personal giants.

My family, Erna, Nora and Friedhelm Beller, for unconditional support at all times. Thank you so much, I love you!

Thomas Kinnen for many discussions and some of the more ingenious solution ideas, for Argus-eyed proof-reading of this thesis, for participating in the interrater reliability study, for eating with me, for walking slowly from time to time and, above all, for being my dearest friend. What a great time we've had!

Fabian Streitel for proof-reading this thesis with unbelievable attention to detail, for participating in the interrater reliability study, and being a really good friend and host. Rock on, k!

Martin Waltl for participating in the interrater reliability study and being a really good friend (besides, thanks for the coffee). Climb to the stars, Martin!

Michael Kanis for participating in the interrater reliability study.

Mika Mantylä, Aalto University, Finland, for releasing a detailed defect classification scheme.

Roland Schulz, member of the GROMACS team, University of Tennessee, Knoxville USA, for proof-reading the parts on GROMACS.

Daniela Steidl for her altruistic sharing of code that allowed me to analyze the complete SVN history of ConQAT.

Benjamin Hummel for Teamscale, for his fruitful and pragmatic ideas and having a great sense of humor.

Elmar Jürgens for providing this challenging and yet very interesting topic, all the discussion and, most important, re-arousing my scientific curiosity.

All the great folks at CQSE GmbH for my daily coffee.

Abstract

Code reviews have become one of the most widely agreed-on best practices for software quality. In a code review, a human reviewer manually assesses program code and denotes quality problems as review findings. With the availability of free review support tools, a number of open-source projects have started to use continuous, mandatory code reviews.

Even so, little empirical research has been conducted to confirm the assumed benefits of such light-weight review processes. Open questions about continuous reviews include: Which defects do reviews solve in practice? Is their focus on functional, or non-functional problems? What is the motivation for changes made in the review process? How can we model the review process to gain a better understanding about its influences and outcomes?

In this thesis, we answer the questions with case studies on two open-source systems which employ continuous code reviews: We find that most changes during reviews are code comments and identifier renamings. At a ratio of 75:25, the majority of changes is non-functional. Most changes come from a review suggestion, and 10% of changes are made without an explicit request from the reviewer. We design and propose a regression model of the influences on reviews. The more impact on the source code an issue had, the more defects need to be fixed during its review. Bug-fixing issues have fewer defects than issues which implement new functionality. Surprisingly, the number of changes does not depend on who was the reviewer.

Contents

1 Introduction

"Twice and thrice over, as they say, good is it to repeat and review what is good." — Plato [PlaBC, 498E]

"A bad review is like baking a cake with all the best ingredients and having someone sit on it." — Danielle Steel

In this section, we motivate our research and give an outline of its structure.

1.1 Motivation

Code reviews have become one of the most widely agreed-on best practices to software quality [CMKC03, BB05, RAT$^+$06]. In a code review, a human reviewer manually assesses program code written and denotes quality problems as review findings. Thanks to the advent of review support tools like Gerrit, Phabricator, Mylyn Reviews, and GitHub [Ger, Pha, Rev, Gitb], a number of open-source projects have started to use continuous, mandatory code reviews to ensure code quality.

However, little empirical research has been conducted to confirm the assumed benefits of the many proposed theoretical review processes [KK09]. Open questions about continuous reviews include: Which defects do reviews solve in practice? Is their focus on functional, or non-functional problems? Why are changes made in the review process? How can we model the review process to gain a better understanding about its dependencies and influences?

In this thesis, we perform two real-world case studies on open-source software systems that employ continuous code reviews.

1.2 Introduction of Research Questions

In the past, research on reviews was mostly constructional in the sense that it suggested new review methods or processes [Mey08, BMG10, CLR$^+$02, WF84]. However, real-world evaluation of code reviews were seldom, and if they were performed, they often counted the review findings [Bak97, Mül04, CW00, AGDS07, KP09a, WRBM97a], but neglected their contextual information.Instead of merely counting the number of review findings, we could generate an in-depth understanding of the review benefits by studying which types of defects were removed. This leads to our first research question:

RQ 1 *Which types of defects do continuous reviews in open-source software systems remove?*

A large-scale survey with developers at Microsoft [BB13] found that the expectations imposed on code reviews and their actual outcomes differ greatly: Programmers think they do code reviews in order to fix functional defects. However, reality shows most code review findings are about non-functional or low-level functional aspects [ML09]. Our second research question captures whether this observation also holds for our OSS systems.

RQ 2 *What is the distribution between non-functional and functional defects?*

An inherent property of changes in the review process—be they functional or non-functional—is the motivation why they were made. Literature on code reviews has contented itself with the diagnosis that some review comments are "false positives". However, this is only an incomplete assessment of the motivation for a change, leaving out changes the author performs without explicitly being told to do. This research question—to the best of our knowledge—is novel in research on code reviews:

RQ 3 *What is the motivation for changes during code review?*

The lack of concrete knowledge about reviews makes it difficult for a project manager to estimate how much effort and how much rework has to go into an issue once it hits review phase. Some projects allow only reviewed code to be passed on into production. An inspection of the factors that impact review outcomes would therefore help project management and project planning, especially in an environment where continuous code reviews are compulsory.

Intuition suggests a number of influencing factors on the review outcome, which include: The code churn of the original code, the reviewer, and the author. Additionally, we assume that to develop a new feature involves writing more new code than to fix a bug. Thus, we would expect an increased number of review findings for new features, and a decreased number for bugfixes.

RQ 4 *Of which kind and how strong are influences on the outcome of a review?*

1.3 Outline

We start our research with a common set of fundamental definitions and conventions about code reviews. We give an overview over the most important works on reviews, and how they relate to our research. Next, we describe the study objects used for research questions 1 to 3, ConQAT and GROMACS. In our first case study, we analyse the types of defects fixed in both systems, and the motivation for their elimination. For our second case study, we build a model that captures influencing factors on the review process. We evaluate this model on ConQAT. Concluding our work, we give an overview of our contributions to research on reviews. Finally, we propose interesting research questions, that resulted from this thesis, for future work in the area of code reviews.

2 Fundamentals

In this chapter, we provide a common terminology of review-related concepts that holds for the rest of this thesis. First, we give some short general naming conventions, then we define the review process in detail.

2.1 Short Terms and Definitions

CMS (Change Management System) A software system to collect and administer modification tasks in the form of issues. Common examples are Bugzilla [Bug], Redmine [Red], Jira [Jir], or Trac [Pro].

Issue The entity in which tasks are stored in a CMS. Other common names are Change Request (CR), and Ticket.

OSS (Open-Source Software) "Computer software with its source code made available and licensed with a license in which the copyright holder provides the rights to study, change and distribute the software to anyone and for any purpose." [Lau08]

SLOC (Source Lines Of Code) A software metric which counts the number of program statement-containing lines, i.e. the lines of code minus blank lines.

VCS (Version Control System) A software system where typically source code is stored in a repository with a retrievable history. Common examples are CVS [Sys], SVN [Sub], Perforce [Per], Mercurial [Mer], or Git [Gita].

2.2 Review Process

In this section we define and model our understanding of a review process. Although a review can take place on any artefact, we define it for the scope of this thesis only on source code.

From the black box view depicted in figure 2.1, a review is a process that takes as input **an original code version** and outputs **a resulting code version**. The **author** is the person responsible for the implementation of the assigned issue. The **reviewer** assures that the implementation meets the quality standards of the project. The original code is a work solely of the original author, whereas in the resulting version the author has incorporated all suggestions from the reviewer so that both are satisfied with the result.

The review process is organized in rounds, cf. figure 2.2: Every review round takes as input **reviewable code.** This is code that the original author deems fit for review. In the first review round, the **reviewable code** equals the original code. Then, the reviewer performs the actual code review, supported by the project's reviewing checklists, style guides

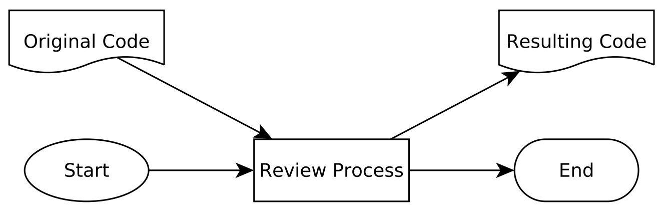


Figure 2.1: The Review Process for source code from an artefact-centred black box viewpoint. Its input is the original code, and its output—possibly altered—resulting code. The state of the issue the code change was part of changes in the process. The diagram uses standard Flowchart semantics.

and tools. An outcome of this is the **reviewed code**, which includes the reviewer's suggestions. These may be stored together with the code, or separate from it. A **review round** is defined as the sequence of the "Perform Review" process followed by either the "Close Review" or the "Integrate Review" process: The number of times the yellow-marked process is executed in sequence with one of the blue processes in figure 2.2 is a counter for the number of review rounds.

If the code fulfilled the quality criteria, the reviewer closes the review process. In this case the resulting code equals the reviewed code from the last review round.

If the code did not meet all quality acceptance criteria, the author is supplied with the reviewed code for rework. By addressing the reviewer's suggestions in the reviewed code, he makes alterations to the code so that he produces again a reviewable code version. The review process begins anew.

Defect

A **defect** is the logical entity describing a number of related textual changes, possibly across files. A defect can be categorized according to the defect topology presented in the following section. We use the term **change** synonymous to defect.

Defect Topology

A change (or defect) can have implications in the form of a functional alteration in the software system, in which case it is a **functional** defect. If it has none, it is a non-functional **evolvability** change. We refine each of these two top-level categorizations further into several sub-groups, cf. figure 2.3. **Structure** defects address problems that alter the compilation result of the code. They represent the most difficult to find defect category, as they require a deep understanding of the system under review. **Visual Representation** defects contain all code-formatting issues without an effect on the compilation result. **Documentation** means problems that are present in program text which has documentary character, like comments and names. A detailed description of the subgroups is given in appendix A. [EW98] elaborates on the sub-categories of the functional defects.

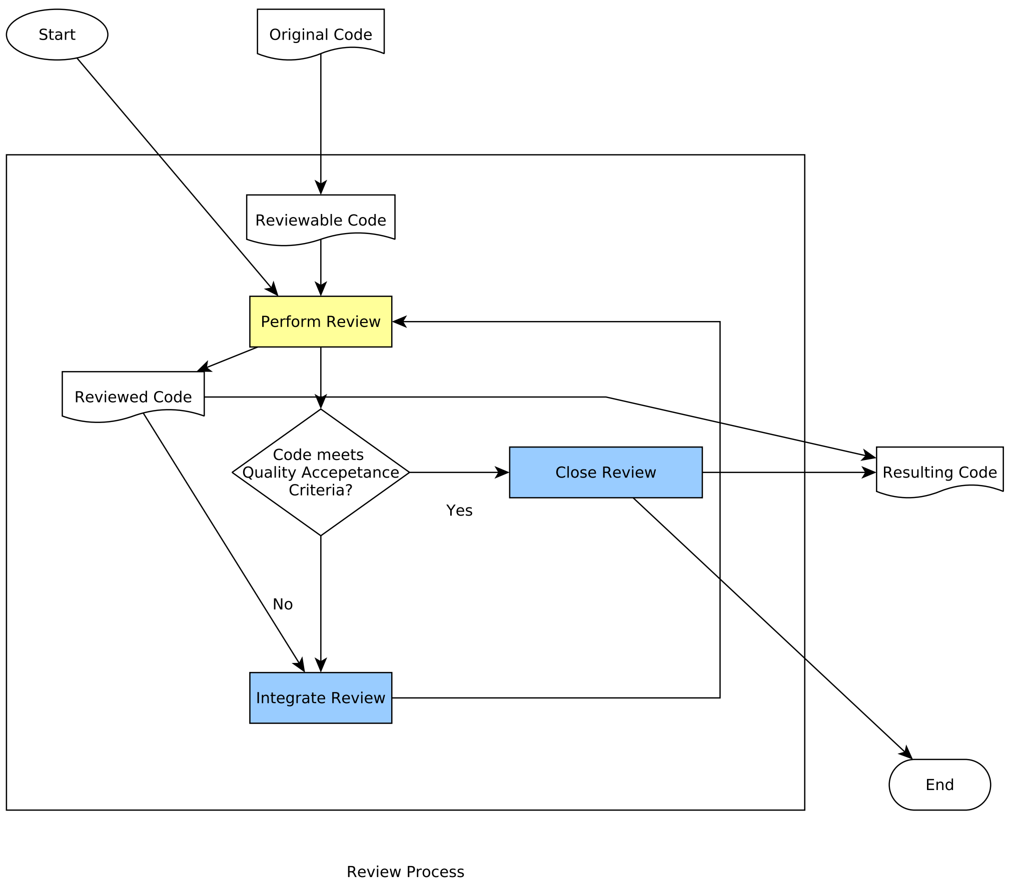


Figure 2.2: A detailed description of the review process. The diagram uses standard flowchart semantics.

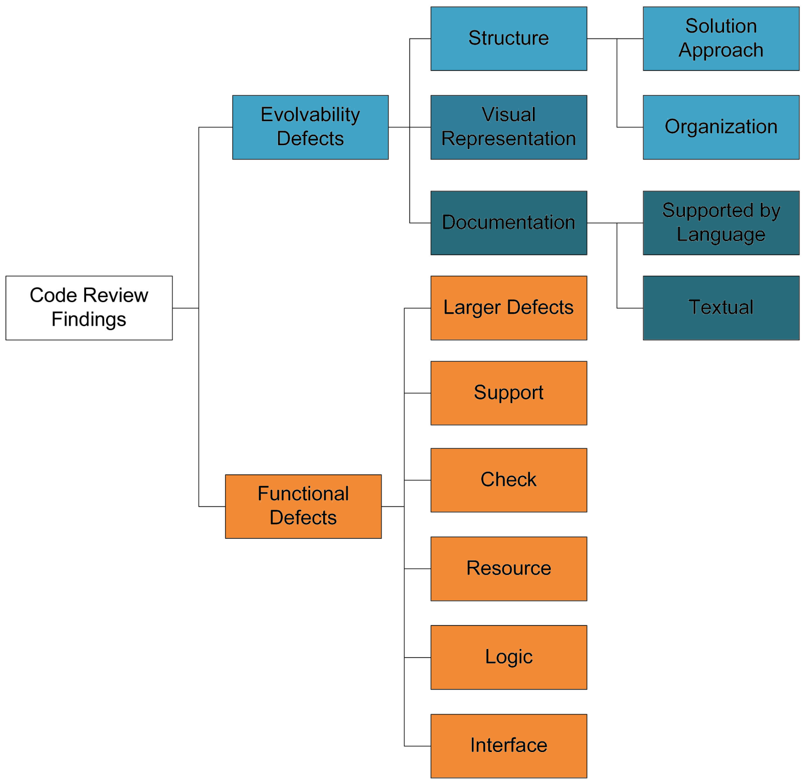


Figure 2.3: The Defect Classification Topology, an adoption from [ML09].

Motivation of a Change

The motivation to remove a defect can either be a comment by the reviewer or a **self-motivated** idea, meaning that the author made a change without a referring review comment. A review comment might not be addressed for various reasons: It could be wrong, too time-consuming to correct, or of doubtful benefit. This is a **discarded**, or, since the author and the reviewer both have to agree to skip the change, an **agreed discarded** change. [ML09] calls these false positives. We believe that the term "agreed discarded" is more suitable, as there are many other reasons to neglect a review comment besides its being incorrect. Additionally, the term false positive does not convey the notion that both author and reviewer have to agree to disregard a change.

3 Related Work

In this chapter we give an overview over prior works which this thesis builds upon.

3.1 A Short History on Reviews

Code reviews first became subject to scientific examination with Fagan's famous 1976 paper on formal inspections [Fag76]. But the idea to perform code reviews is even older and dates back to the first programming pioneers like von Neumann: They considered review such an essential part of their programming routine that they did not even mention it explicitly [KM93]. Following Fagan's groundbreaking paper in 1976, a whole subdiscipline of Software Engineering dedicated itself to the topic of reviews. The discipline is not fixed on investigating code reviews, but performs research on reviews of other software engineering artefacts like requirement documents and architectural designs [GW06, MRZ^{+}05]. [KK09] gives an overview over the past and the status quo of research on reviews. They note a lack of empirical knowledge on code reviews, and suggest that more such studies be conducted to measure the effects of different theoretical review suggestions. With our work on quantifying code reviews, we follow their call.

3.2 Formal Inspections

Two categories of code reviews have established themselves over the course of the last four decades of research on reviews. Heavy-weight Fagan-style inspections, and light-weight code reviews with an emphasis on productivity. More formal review processes tend to be called inspection, whereas the other processes are referred to as reviews. While [KK09] notes some authors try to avoid the term "inspection" and use "peer reviews" instead, they find no fundamental difference in work on reviews compared to work on inspections. Orthogonal to this classification, reviews with two participants are sometimes called "pair reviews", and reviews with more participants "circle reviews" [WYCL08].

The Fagan inspection mandates a waterfall-like process to develop software, where review and rework phases are imperative at the end of pre-defined stages like design, coding, or testing. An inspection team comprises four roles, a moderator, a designer, an implementer, and a tester. The Fagan inspection begins with an overview phase, the initial team gathering, in which the designer describes the system parts to be inspected. He also hands out code listings and design documents. After that, the team shall meet in "inspection sessions of no more than two hours at a time" [Fag76], where it discusses errors that the participants found in the preceding individual preparation. The review meeting's sole purpose is to uncover deficiencies, not to correct them. After the inspection session, the author has to resolve all the uncovered errors from the meeting in the rework phase. In a follow-up, either the moderator or the whole team—depending on the number and impact of the changes—makes sure the rework fixes the addressed problems.

3.3 Light-Weight Reviews

Although an initial success with both the research and practitioner community, Fagan inspections have several disadvantages that somewhat hindered their continuous and widespread use across organizations: They mandate a plethora of formal requirements, most notably a fixed, formal reviewing process that does not adapt well to agile development methods [Mar03]. It made Fagan inspections lengthy and inefficient. Several studies have shown that review meetings do not improve defect finding [Vot93, MWR98, BLV01, SKI04]. Only one study reported contrary results, stating that review meetings did improve software quality [EPSK01]. As a result, the research community developed more light-weight, adhoc code reviewing processes that better suited environments where test-driven and iterative development take place [DHJS11, UNMM06, MDL87, Mey08, Bak97, BMG10].

Light-weight review processes are characterised by fewer formal requirements, a tendency to include tool support, and the overall strive to make reviews more efficient and less time-consuming. These advances allowed many organizations to switch from an occasional to a mandatory, continuous employment of reviews. [Mey08] describe their experiences with continuous light-weight reviews in a development group comprising three globally distributed programmer teams. Light-weight reviews often leave out the team meeting, and reduce the number of people involved in the review process to one reviewer. Adversely, [WRBM97b] found that to exploit the full effect from reviews, the optimal number of reviewers should be two. In some light-weight processes the author and reviewer may switch roles, or replace the "asynchronous review process" with a pair programming session [DHJS11]. In stark violation of the rules of the Fagan inspection, reviewers in some light-weight processes may make changes to the code themselves. This is the case in both of our study objects, ConQAT and GROMACS which employ light-weight review processes.

3.4 Review Effectiveness and Efficiency

Fagan provided data on inspection rates, i.e. how many SLOC without comments could be reviewed in one hour [Fag76]. The reported values varied greatly from 898 to 130 SLOC. [KP09b] performed an extensive case study on the review rate. They found that a review rate of 200 LOC/hour or less was an effective rate for individual reviews. Their research concentrated on functional defects and did not address maintainability defects. Given a reported 75% of defects in reviews are non-functional [ML09], it stands to question whether the results are applicable to a modern review process.

Sauer et al. [SJLY00] argue that individual expertise is the most important factor in review effectiveness, and Hatton [Hat08] supports this claim: In his experiment, he found stark differences in the defect finding task among individual reviewers.

The ability to understand source code and perform reviews is called "software reading" [CLR$^+$02]. The initial idea for this came from [PV94], who advocated scenario-based reading. Instead of generic checklists, the scenarios shall provide reviewers with more accurate instructions for their review. Several code reading techniques like Defect-Based Reading, Perspective-Based Reading, Object-Oriented Reading, or Use-Based Reading have been suggested to educate code readers [WYCL08, CLR+02]. [EW98] depicts a code review process based on classic standard checklists. They provide an exemplary checklist for reference. GROMACS uses a small checklist, while ConQAT has no such document. Reviewers in either system were not aware of code reading.

Code reading can be seen as an inspection without meetings, introducing light-weight reviews. [WRBM97b] compare the code reading technique stepwise abstraction to functional and structural testing, a replication study performed "at least four times [...] over the last 20 years". Step-wise abstraction means for the reviewer to build a specification from the code, and then to compare it to the official specification that the code was developed from. Their findings are that the three techniques are similar with regard to finding defects, and that they are best used in combination.

3.5 Comparison With Other Defect Detection Methodologies

Here we describe research that compares the effectiveness of code reviews to detect functional defects in a program with that of other quality enhancing techniques, namely testing and pair programming. [KK09] notes that in the past decade, research on reviews has increasingly taken to developer surveys. In light of this, [BB13] conducted a survey on developers at Microsoft, inquiring developers' motivation to do reviews. Their main expectation was to fix functional defects, but really failure-related comments make up only a small proportion of the corrected defects [ML09, BB13]. Our results show this is also the case for ConQAT and GROMACS.

Code Reviews And Testing

Several researchers tried to measure how effective code reviews were in detecting program faults, and compared them to structural and functional program testing [KK09]. Figure 3.1 shows an overview of the effectiveness researchers have measured. There seems no consensus whether testing or reviewing is more effective: Three papers favoured testing, four were indifferent, and two favoured inspection. The question whether testing or reviewing is more efficient received similar diverse answers across papers: Two found testing to be more efficient, and one inspections. The reported error detection numbers are surprisingly low, at an average of 0.68 defects per hour for inspections, and 0.10 for testing [KK09]. [RAT+06] states that "absolute levels of effectiveness of defect detection techniques are remarkably low" and that "on average, more than half the defects remain".

Code Reviews and Pair Programming

Like code reviews, pair programming is often attributed with higher code quality, fewer defects, shorter development times, and other beneficial outcomes when compared to individual programming [BA04, WKCJ00]. Because the two are believed so similar, a separate review can be replaced by a pair programming session according to some review

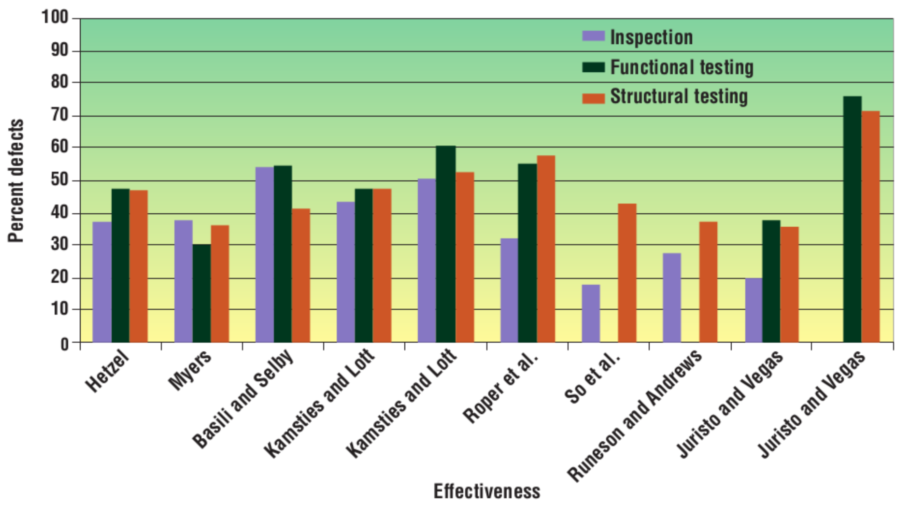


Figure 3.1: Comparison of the results of different papers that compare review with test effectiveness (Source: [RAT+06]).

processes, for example in [DHJS11]. [CW00] contradict this statement, claiming that pair-programming in their example "simply worked better" than "cod[ing] individually for awhile, and then review[ing] the changes with their partner".

Müller [Mül04] investigates whether reviews are an alternative to pair programming. In his paper, he compares pair programming to individual programming plus a succeeding review. He argues that simply by knowing that a review will follow, the author produces better programs. His findings are that reviews can compete with pair programming in terms of reliability, at a fraction of the costs of pair programming. [Mül05] tries to evaluate the claims further in two controlled examples with university students.

In a controlled experiment with 295 Java developers [AGDS07] evaluated whether these hypotheses hold in practice. Their results show that pair programming does not reduce the time to finish a programming task correctly, nor does it increase the proportion of correct solution. However, pair programming has a significant 84% increase in the combined man hours necessary to produce a correct solutions. These results question the practice of replacing reviews by pair programming.

3.6 Supporting Tools

With the advent of light-weight review processes arose a need for supporting software tools, preferably integrated into the development IDE [CdSH+03]. [BMG10] introduced one such tool in 2010 for the Eclipse IDE, ReviewClipse, now Mylyn Reviews [Rev]. Their idea is for reviewers to perform reviews on a commit directly after it has been pushed into the VCS. ReviewClipse automatically creates a new "review process", assigns a fitting reviewer, and opens a compare viewer for this commit.

A popular review tool is the OSS Gerrit [Ger], started in 2008 by Google. Gerrit "is a web based code review system, facilitating online code reviews for projects using [...] Git."

Gerrit supports management of the review process with change tickets, and the review itself with an interactive side-by-side comparison of the old and new code versions. It allows any reviewer to add inline comments in his web browser. Reviewable code for a ticket is saved in so-called patch sets. A review round takes place on one patch set. Gerrit accommodates the postulate of [WRBM97b] for more circular reviews by encouraging many reviewers in one ticket. A key feature of Gerrit is that it integrates the changes into the main repository only after the reviewer expressed his consent to it [Mil13]. In practice, Gerrit is often used in combination with Jenkins [Jen] because it enables automatic build and test verification of the changes [Mil13]. This could be considered a first automated review. Should it fail, no manual reviewer needs to read the error-causing code, which increases review efficiency in the spirit of light-weight reviews. All the described procedures are part of GROMACS's review practice, which uses Gerrit. ConQAT has no review management tool.

For its closed-source projects, Google uses Mondrian, a company-intern tool that was the trigger for Gerrit's development. It is similar to Gerrit but highly tailored towards Google's development infrastructure [Ken06].

Phabricator is Facebook's open-sourced tool support for reviews [Pha], developed since 2000 and publicly released in 2011. Github's review system works with pull requests, which comprise the code, a referenced issue and possibly review comments [Gitb, Ent]. It is available freely for OSS since 2008. Both Phabricator and Github's review system are web-based and very similar to Gerrit.

Microsoft developed and deployed its own code review tool called CodeFlow since 2011 [BB13]. It offers a unique synchronous collaboration possibility between the author and the reviewer, as they can work on the review at the same time thanks to an integrated live chat.

3.7 Defect Topologies

Computer scientists have produced an abundance of defect classifications over the years [Wag08], eventually leading to an IEEE standard in 1993 [53910]. Figure 3.2 visualizes the development of the different defect topologies. The IEEE standard and its draft were the basis for two classifications by IBM and HP. Researchers at IBM invented the Orthogonal Defect Classification (ODC) [CBC^{+}92]. They classify a defect across six orthogonal dimensions, the first of which is the defect type. The defect type is further refined into eight categories. HP suggested a similar classification across three dimensions, called "Defect Origins, Types, and Modes" [Gra92].

Case studies evaluated these topologies and found they are difficult to use [WJKT05, DM03]. This is because they are too general to be helpful and need bespoke tailoring before they can be used in practice.

Consequently, researchers refined these topologies. [EW98] have shown that their model, based on IBM's ODC and with influences from [Hum95], had high interrater reliability. Mäntylä and Lassenius—in search of a defect topology for review findings—based their topology largely on this empirically validated classification scheme [ML09].

Our own classification builds upon these works, and makes small adjustments to the evolvability and functional defect sub-categories. Moreover, we removed the false positive top-level category because the motivation for a change is an orthogonal categorization to its type. The fundamental difference is that we classify changes, and not review comments with our topology. [ML09] research which type of defects code reviews find, and what the distribution between evolvability and functional defects is. Our research is confirmatory regarding these questions—except for a different understanding of defect—and additionally examines the motivation for changes. Moreover, we build a generalised linear model on the influences of code reviews. Other researchers in software engineering have already used generalised or mixed-models, but not to build a model on the influence of reviews [AGDS07].

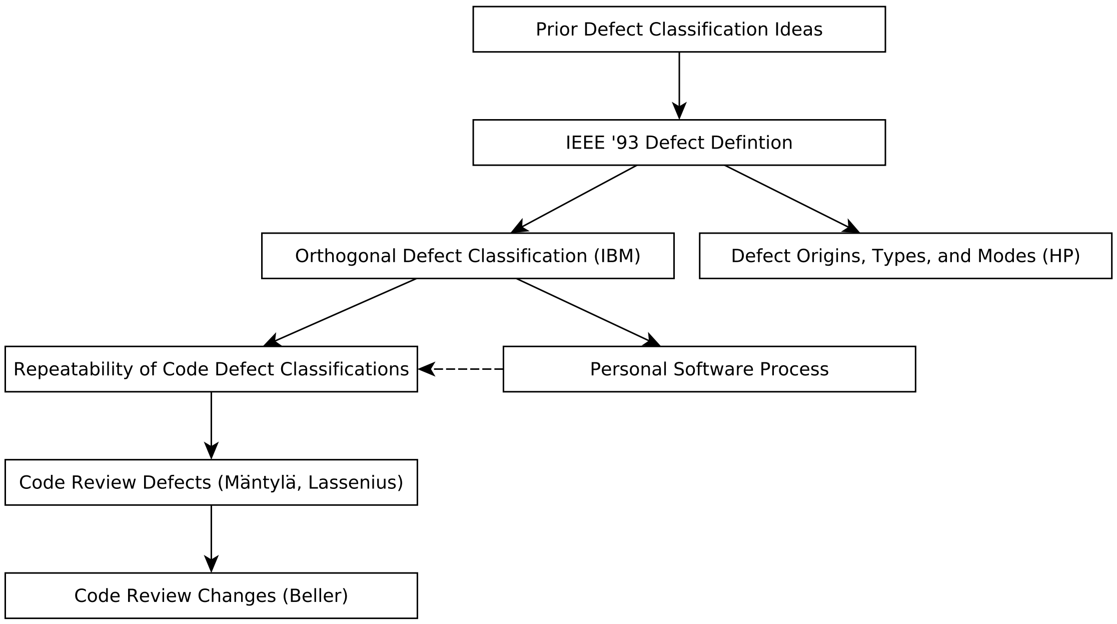

Prior Defect Classification Ideas
IEEE '93 Defect Defintion
Orthogonal Defect Classification (IBM)
Defect Origins, Types, and Modes (HP)
Repeatability of Code Defect Classifications
Personal Software Process
Code Review Defects (Mäntylä, Lassenius)
Code Review Changes (Beller)

Figure 3.2: The development of code defect classifications.

4 Study Objects: ConQAT and GROMACS

This chapter gives an overview over the OSS systems that we evaluated in our case studies in chapters 5 and 6. Table 4.1 gives a short overview of the key metrics of both systems.

4.1 ConQAT

ConQAT is "an integrated toolkit for creating quality dashboards that allow to continuously monitor quality characteristics of software systems. [...] ConQAT is an open-source project under the Apache 2.0 license and the frequent releases are available as free downloads." [Con] ConQAT is mostly written in Java and uses ant as a build tool. On January the 30th 2013, it consisted of 4,345 files with a total of 496,404 LOC (260,465 SLOC).

History

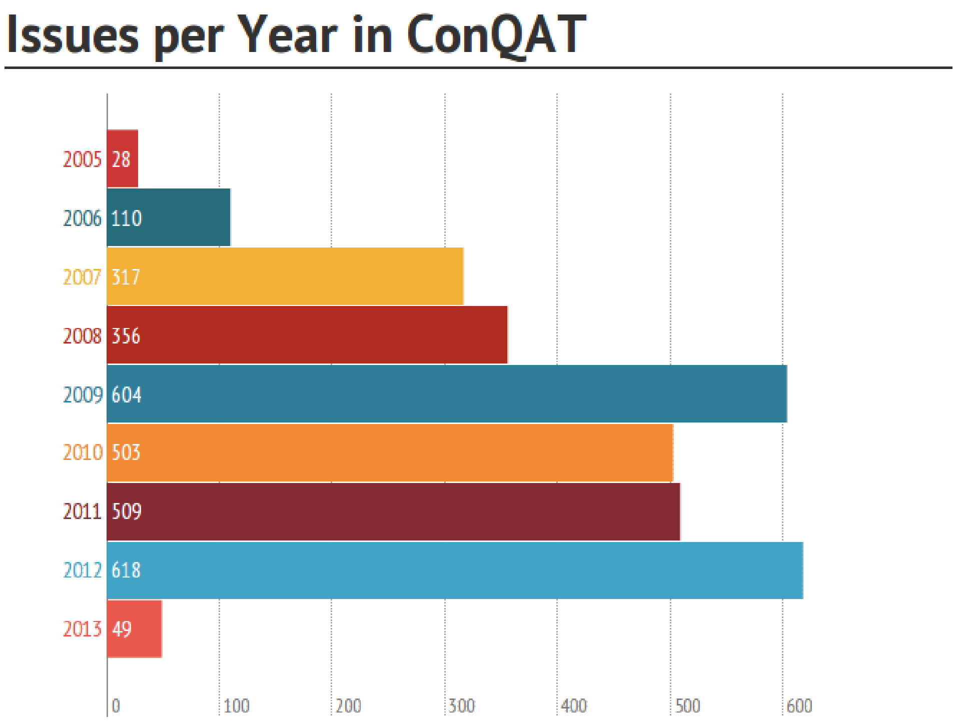


Figure 4.1: The number of issues created per year. Data goes until 30th of January 2013. The total number of created issues is 3094.

ConQAT was originally developed and hosted at TU München. While the precise start of the project is unknown, ConQAT release 1.0 shows August the 7th 2007 as its timestamp. A repository analysis of the VCS dates the first commit to June 17th 2004. The eldest change request in the CMS, then Bugzilla, dates back to 2005. Figure 4.1 shows the project activity as the created number of issues in Redmine per year. In the ConQAT source files the copyright header states "Copyright 2005-2011 The ConQAT Project", also suggesting an official start of the project in 2005.

As of 2012, CQSE GmbH [Gmb], a university startup founded by the core developers of ConQAT, has continued to host, maintain and develop ConQAT, with only minor contributions from externals. There is now a separation between the OSS repository and a closed source part accessible exclusively to CQSE employees. In this thesis, we have concentrated on the OSS part, comprising eight years of development history.

Developers

Because of its history as a university project, ConQAT was subject to many different research ideas and authors with strongly diverging backgrounds. Parts of ConQAT were written during three week phases of mandatory university course projects by a small group of participating students ($n < 15$). Even the usual development cycle of the core developers saw phases of high workload combined with periods when almost no work on ConQAT took place. Overall, contributors ranged from first-year students with little programming experience in their early 20s to senior Java developers with a doctorate degree and 15 years of programming experience. It was therefore mandatory to set up a development process in which the different backgrounds of the developers and the changing load of development activities would not lead to maintenance issues. To counter these problems, Deißenböck et al. [DHJS11] invented the LEvD process (cf. section 4.1).

On January 30th 2013, there were 13 active contributors at CQSE GmbH, most of which were not full-time developers and only seldomly commited at all. The CMS lists 15 active users, and 185 users in total. An SVN repository analysis yields 52 committers in total, with a very uneven distribution of commits (minimum 1, maximum 791), cf. figure 4.2: There is a clear separation between the main developers and sporadic contributors.

Tools

ConQAT uses Redmine as its CMS, and SVN as its VCS. There is no external tool support for code reviews. Reviewers use a small plugin called "RateClipse", which displays the review status of a file. Reviewers perform their work directly in the code in Eclipse, cf. figure 4.3.

Review Process—The LEvD Process

The Lean Evolution and Development Process (LEvD) is ConQAT's light-weight review process, which defines only two roles: An author and a reviewer.

> The LEvD-Process is intended for an environment with small to medium-sized teams with software maintenance, enhancement, and development tasks

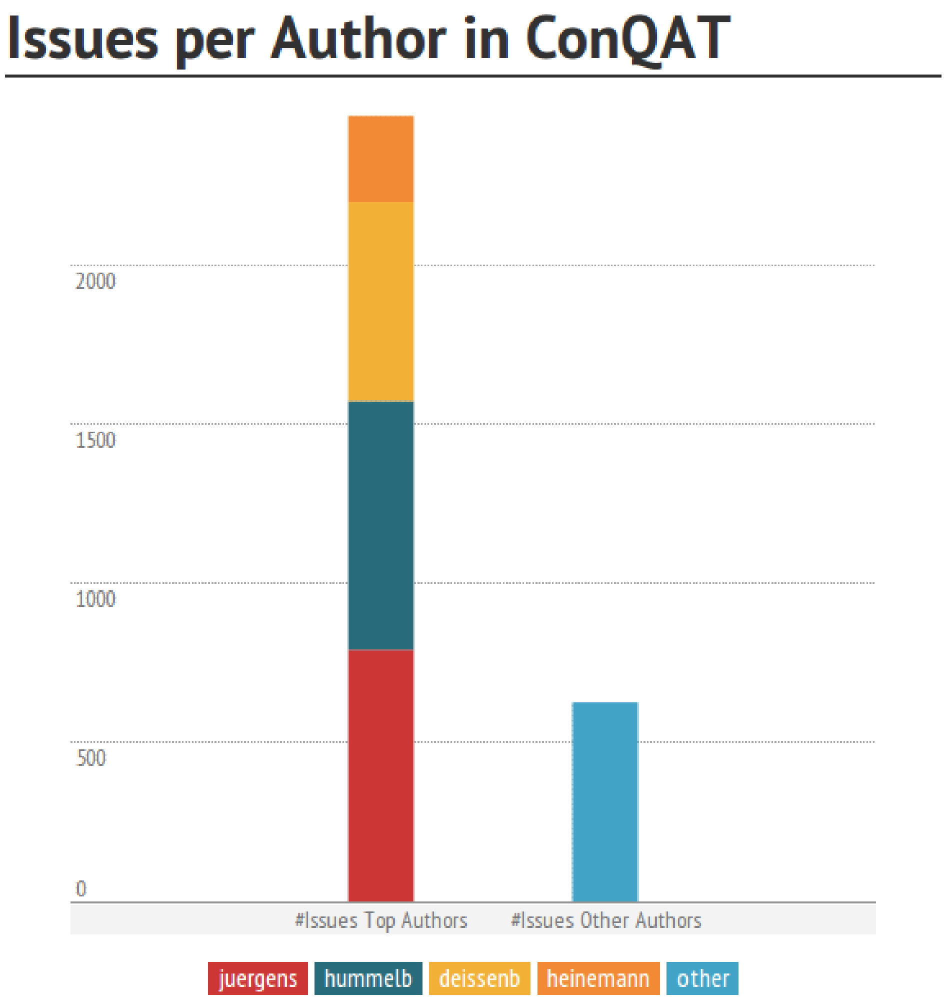
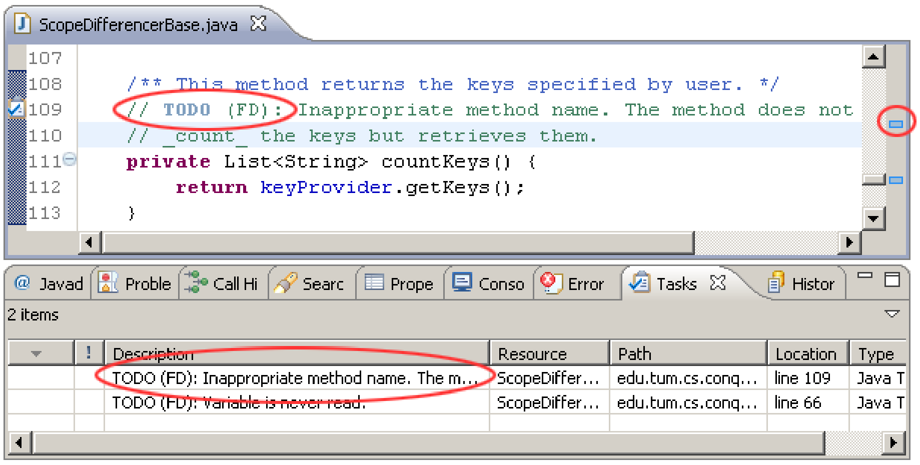


Figure 4.2: The number of issues assigned per author. Only four authors are responsible for over 75% of the issues. This is ConQAT's core developer team.

> and a fairly high rate of fluctuation. Therefore the process concentrates on code quality and traceability of activities. — [DHJS11]

The development process in ConQAT is strictly issue-based: LEvD mandates that every commit to the VCS contains the ID of the change request this commit belongs to, cf. listing 4.1. This way it is possible to link changes in the VCS to the issue in the CMS. For a change to go into the repository, the author has to create an issue in the CMS and the reviewer must close it according to the following process.



```
    /** This method returns the keys specified by user. */
    // TODO (FD): Inappropriate method name. The method does not
    // _count_ the keys but retrieves them.
    private List<String> countKeys() {
        return keyProvider.getKeys();
    }
```

	!	Description	Resource	Path	Location	Type
		TODO (FD): Inappropriate method name. The m…	ScopeDiffer…	edu.tum.cs.conq…	line 109	Java T
		TODO (FD): Variable is never read.	ScopeDiffer…	edu.tum.cs.conq…	line 66	Java T

Figure 4.3: Appearance of review findings as TODOs in the Eclipse IDE (Source: [Dei09]).

Listing 4.1: A commit message in ConQAT, referencing change request 4521.

```
CR#4521:
Refactor common code into base class
```

A reviewer writes his findings directly into the source code file as program comments, and then commits the changed file to the VCS [Dei09]. If a finding represents a more high-level defect (i.e. the author forgot to commit files, a feature does not work or is incomplete, etc.), a note is made in the CMS instead.

A central idea of LeVD is that the reviewer assesses not only the changes, but the whole file in which the changes took place. This is a fundamental difference to other review processes, where only the changeset is reviewed—e.g. in Gerrit, cf. section 3.6.

In the LeVD model, every artefact under quality control is in one of three states at any given time: Red, Yellow, or Green.

> “By default, a newly created artifact is rated RED. The author of an artifact can change its state to YELLOW to express that he is confident that all quality requirements are met. With this color change, the author signals that the artifact is ready to be reviewed. A reviewer, other than [t]he authors, performs a quality review of the artifact and rates it GREEN if all quality requirements are met or RED if one ore more requirements are violated. [...] If the reviewer rated the artifacts RED, the author corrects the quality deficiencies and rates the artifact yellow, when he is finished. A GREEN artifact is automatically rated RED

> if it is subject to any modification. This way, it is ensured that all modifications are properly reviewed." [Dei09]

Figure 4.4 depicts this process. Two deviations from the process have established themselves in practice: If the author and the reviewer work together as pair programmers, they may rate an artefact directly green, omitting the review phase. Additionally, if the reviewer finds obvious minor defects—a typo in a variable name or a small problem in the comment—he may alter them without consent from the author. Figure 4.5 shows an example of this.

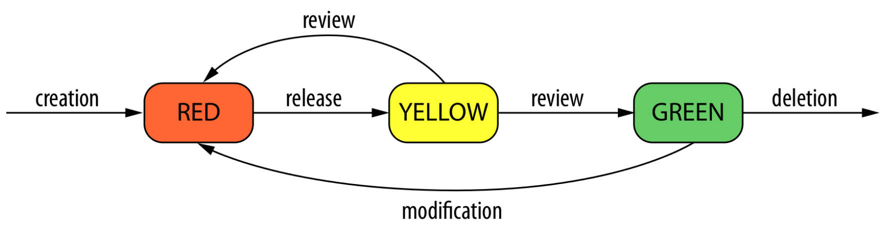
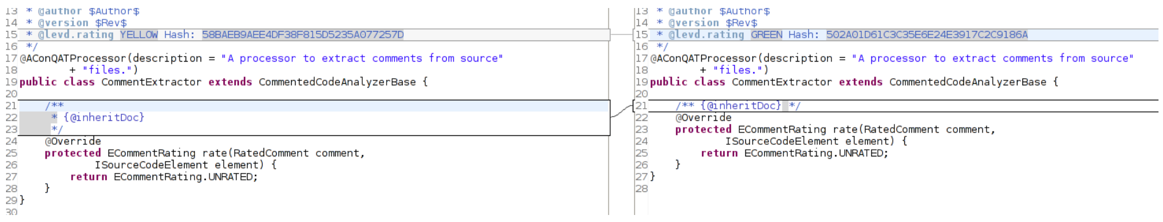


Figure 4.4: The LEvD process showing the different states an artefact can be in (Source: [Dei09]).



```
 * @author $Author$
 * @version $Rev$
 * @levd.rating YELLOW Hash: 58BAEB9AEE4DF38F815D5235A077257D
 */
@AConQATProcessor(description = "A processor to extract comments from source"
        + "files.")
public class CommentExtractor extends CommentedCodeAnalyzerBase {

    /**
     * {@inheritDoc}
     */
    @Override
    protected ECommentRating rate(RatedComment comment,
            ISourceCodeElement element) {
        return ECommentRating.UNRATED;
    }
}
```

```
 * @author $Author$
 * @version $Rev$
 * @levd.rating GREEN Hash: 502A01D61C3C35E6E24E3917C2C9186A
 */
@AConQATProcessor(description = "A processor to extract comments from source"
        + "files.")
public class CommentExtractor extends CommentedCodeAnalyzerBase {

    /** {@inheritDoc} */
    @Override
    protected ECommentRating rate(RatedComment comment,
            ISourceCodeElement element) {
        return ECommentRating.UNRATED;
    }
}
```

Figure 4.5: A trivial change corrected by the reviewer: He removes blank lines in the `rate` method's JavaDoc.

4.2 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." [Gro] GROMACS is an OSS project released under the GNU LPGL. Its primary language is C. In July 2013, [Ohl] reported 1,449,440 SLOC in C for GROMACS.

History

"GROMACS was first developed in Herman Berendsen's group, department of Biophysical Chemistry of Groningen University" [Gro]. According to [Ohl], the first data point in the VCS dates back to November 1997, but the project already had 74,625 SLOC in C then, indicating a prior start of the project. The first mention of GROMACS in a scientific paper dates back to 1995 [BvdSvD95]. The reported 1,449,440 SLOC in C for GROMACS amount

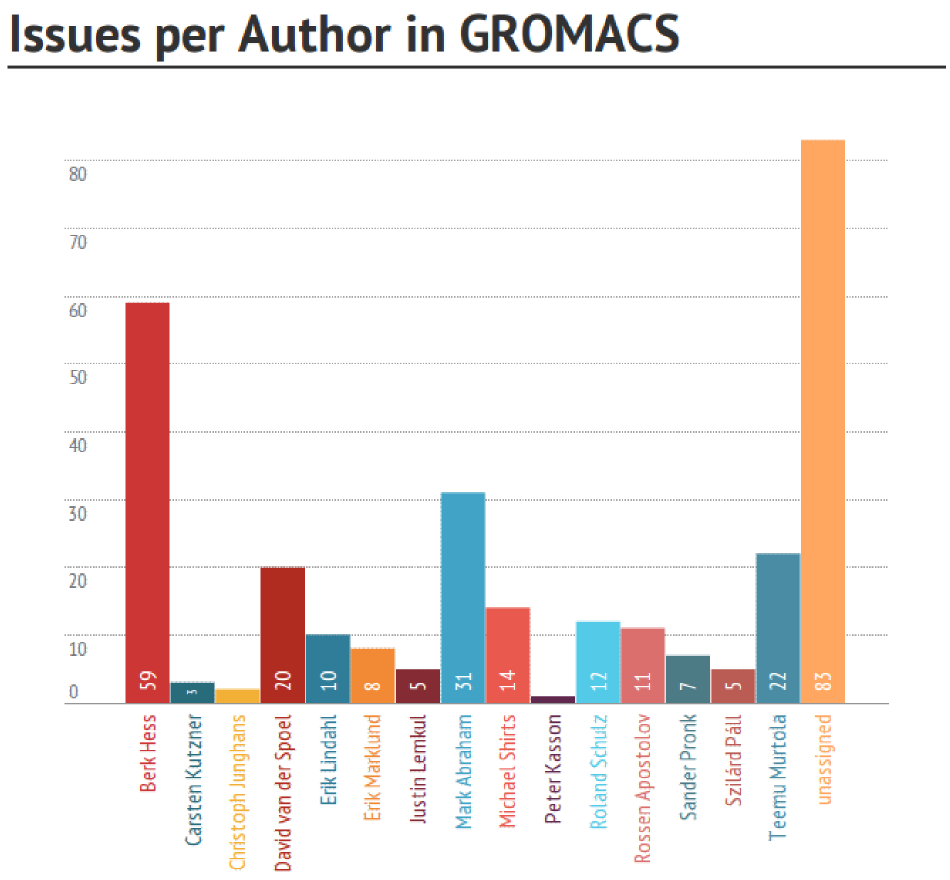


Figure 4.6: The number of issues assigned per author. Data goes from 1st of November 2011 until 1st of July 2013.

to ∼80% of the total code. Other languages with a significant amount of code include only Fortran (∼8% of the total code) and C++ (∼6% of the total code). The use of Gerrit for code review began in August 2011.

Developers

"[GROMACS] is a team effort, with contributions from several current and former developers all over world" [Gro]. The project pages lists three head authors, one development manager and twelve current developers. Four people are listed as "Contributors" and twelve as "Former Developers". [Ohl] states 44 contributors. Figure 4.6 depicts the number of issues per author.

Tools

GROMACS uses Redmine as its CMS, and Git as its VCS. Code reviews are performed in Gerrit.

Review Process

The development process in GROMACS is mostly issue based. For a change to go into the repository, a change request in the CMS has to be created and properly closed according to the following review process: GROMACS requires commits to pass code review in Gerrit before they are allowed to be merged into the VCS. Smaller changes may go in without an explicit change request, but they still need to be reviewed with Gerrit. [Gro] describes the reviewing process:

> 1. `https://gerrit.gromacs.org/#q,status:open,n,z` shows all open changes
> 2. A change needs a +2 review and a +1 verified to be allowed to be submitted. [...]
> 3. A change is submitted by clicking "Submit". This should be done by the reviewer after voting +2. After a patch is submitted it is replicated to the main git server.
>
> Do not review your own code. The point of the policy is that at least two non-authors have voted +1, and that the issues are resolved in the opinion of the person who applies a +2 before a merge. If you have uploaded a minor fix to someone else's patch, use your judgement in whether to vote on the patch +1.

[Gro] lists in its "Guide for reviewing" (spelling mistakes are part of the original):

> - First and foremost, check correctness to the extent possible; As portability and performance are the most important things (after correctness) do check for potential issues;
> - Check adherance to GROMACS coding standards;

- We should try to ensure that commmits that implementing bugfixes (as well as important features and tasks) get a Redmine entry created and linking between the commit the Redmine entry is ensure. The linking is done automatically by Redmine if the commit message contains `keyword #issueID`, the valid syntax is explaned below.
- If the commit is a bugfix:
  - if present in Redmine it has to contain valid reference to the issue;
  - if it's a major bug, there has to be a bug report filed in Redmine (with urgent or immediate priority) and referenced appropriately.
- If the commit is a feature/task implementation:
  - if it's present in Redmine it has to contain valid reference to the issue; [...]

Category	ConQAT	GROMACS
Development time	$\geq$ 8 years	$\geq$ 18 years
Developers	~10 active, ~50 overall	~16 active, ~44 overall
Language	Java	C (mostly)
SLOC	260,465	1,449,440
Code Reviews since	2007	2011
Review mandatory	Yes	Yes
Tool support	RateClipse (Eclipse IDE)	Gerrit
Number of Reviewers	1	≥ 2
Number of Review Rounds	$[1; \infty[$	$[1; \infty[$

Table 4.1: Comparison of ConQAT and GROMACS.

5 Analysis of Defects in Reviews

In this chapter, we conduct a case study on review finding types for two real-world OSS software systems in practice.

5.1 Structure of Case Study

We repeat the three research questions we are answering in this chapter. Additionally, we give a detailed outline of the chapter.

RQ 1 *Which types of defects do continuous reviews in OSS systems remove?*

RQ 2 *What is the distribution between evolvability and functional defects?*

RQ 3 *What is the motivation for changes during code review?*

In this chapter, we analyse which types of defects continuous reviews in two OSS systems identified. We compare the similarities between the different defect distribution profiles created for ConQAT and GROMACS. After abstracting the detailed distribution profile, we determine the ratio between top-level maintenance and functional defects, and put the ratio in context with other studies on different software systems. Next, we focus on how many of the review suggestions were useful in the evolution of the software. To conclude the case study, we identify problems that could threaten the validity of the results and show how we mitigated them. We conducted our case studies based on the guidelines for empirical research in software engineering [KPHR02].

5.2 Types of Review Defects

The first research question deals with the types of defects solved during reviews. Apart from answering the research question, we also elaborate in this section on how we collected the data relevant to all research questions in this thesis. RQs 2 and 3 conduct further research on the data originally collected for RQ 1.

RQ 1 *Which types of defects do continuous, light-weight reviews in OSS systems remove?*

RQ 1 is confirmatory in nature. To answer it, we set up a modified replication of the study performed for the second research question in [ML09]. The important difference between the two studies is that we assess all changes made in the review, whereas [ML09] assess only the review comments denoted by the reviewer.

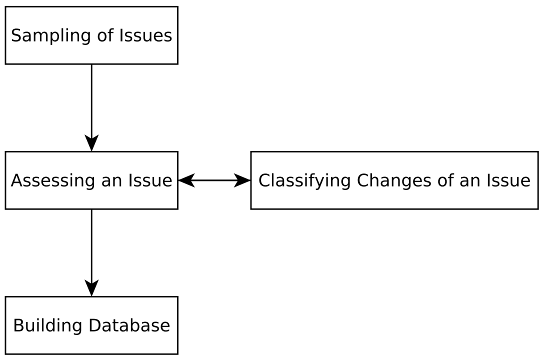


Figure 5.1: Study design of RQ 1.

Study Design

Figure 5.1 depicts the sub-steps of the study design. The following sections describe each of the steps in more detail.

For the evaluation, we chose the projects described in chapter 4. The reason for the selection of ConQAT was that—since we are part of its development team—we have a deep domain-specific knowledge on it. Furthermore, the project has a well-documented history in the VCS and CMS, and uses continuous code reviews.

As ConQAT's counterpart, we chose GROMACS since we wanted to compare two systems that employ different review processes and tools (LeVD and Gerrit, respectively), and because GROMACS had a documented history of performing mandatory code reviews. This holds only for a small set of OSS projects that we could find. Even if they claim to use Gerrit, it is often optional, or only for newcomers.

Sampling of Issues

Since we expected many confining variables (cf. 6), we created two samples from the large ConQAT data set, so that we could compare the two sub-samples later on: The last one hundred issues, and a randomized sample of issues from the population. The one hundred most recent issues are representative of the current development of reviews in ConQAT, whereas the sampled issues should provide an approximation of the general defects uncovered in ConQAT reviews. For GROMACS it was not feasible to establish two sufficiently large sample groups because of a much smaller set of available data points. In total we created three data sets: ConQAT Random, ConQAT (Last) 100, and GROMACS.

Because of the quantity of total relevant issues in ConQAT and GROMACS—over 900 and 250 in ConQAT and GROMACS, respectively—we could not assess all issues. Instead, we selected a representative sample of issues from both systems. All data sets should consist of about ~100 issues. This makes them more comparable among each other. Since we expected the author of an issue to be one of the most dominant influencing factors in reviews, we performed a stratified sampling of issues to guarantee equi-frequent authors.

Assessing an Issue

To collect a data set of review changes, we used the following procedure: First, we selected a representative sample of issues from the CMS of either OSS software. Then we categorized for each issue metadata like the author, reviewer(s) and change type, which was mostly available via the CMS. Finally, we established whether the issue was suitable for inclusion in the study (valid), or unsuitable (invalid). We explain the technical details of this in section 5.2.

If the issue was valid, we could analyse how many review rounds took place. For each review round we categorized the changes that occurred in this round by a manual source code comparison. Additionally, we integrated information from the CMS into the review round analysis.

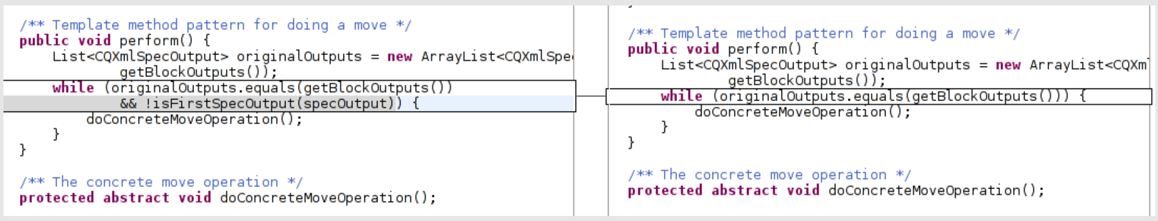
Example 1 Issue 4387 caused a lot of code churn in ConQAT. While the author still reworked parts of the reviewed version, the reviewer began with the review of the already reviewable files to reduce his waiting time. The reviewer and author agreed on this procedure in a note in Redmine. Later, the rest of the code was made reviewable by the author.

Based on the chronology of commits in the SVN, we would have classified this as two rounds, although it is per definitionem only one.

Classifying Changes of an Issue

Additional to changes triggered by review comments, we noticed changes in the code during review rounds which were not based on any of the reviewer's suggestion. It is clear that without a review, these changes would not have been made. Therefore, such changes to the code—be they from the original author or the reviewer—are an outcome of the review process, and should be included in an analysis of the review process. By including them, we hope to capture not only the review findings, but all changes triggered by the review process. Whereas most literature merely classifies the review suggestions—and in some studies like [ML09] also whether these were realised, or discarded (false positive)—we base our type classification on a comparison of the actual changes from the reviewable code at round i and compare it to the reviewable code of the prior round $i - 1$. This way, we consider all changes that happened in-between an outcome of the review.

Example 2 A self-motivated, functional change of the code by the author within a review round.



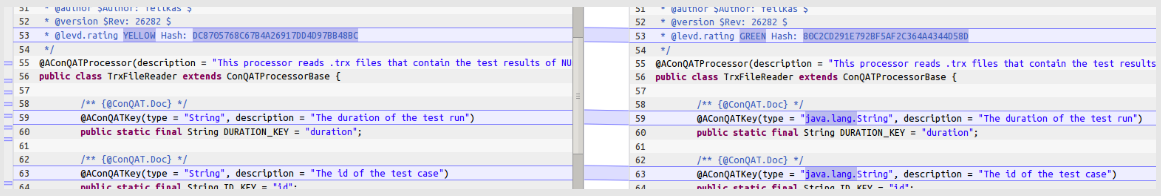
```
/** Template method pattern for doing a move */
public void perform() {
    List<CQXmlSpecOutput> originalOutputs = new ArrayList<CQXmlSpe
            getBlockOutputs());
    while (originalOutputs.equals(getBlockOutputs())
            && !isFirstSpecOutput(specOutput)) {
        doConcreteMoveOperation();
    }
}

/** The concrete move operation */
protected abstract void doConcreteMoveOperation();
```

```
/** Template method pattern for doing a move */
public void perform() {
    List<CQXmlSpecOutput> originalOutputs = new ArrayList<CQXm
            getBlockOutputs());
    while (originalOutputs.equals(getBlockOutputs())) {
        doConcreteMoveOperation();
    }
}

/** The concrete move operation */
protected abstract void doConcreteMoveOperation();
```

To accommodate for this, we use an adapted version of the defect classification originally published in [ML09], cf. appendix A. The differences are minor: We included some clarifications on how to rate certain Java-specific language constructs, and we removed sub-categories in the resource defect category because we expected these defects to be so few that further separation would not increase precision. Most important, we removed the false positive category. We find it is an orthogonal concept to the type of a change: Per definitionem, either a code change happened, and then we can categorize this change in the appropriate category. Or no code change occurred, but then it is also not a false positive.

In LEvD, reviewers may introduce trivial code changes in the reviewed code (cf. figure 2.2). While this is technically not possible in Gerrit, the reviewer can switch roles with the author and commit a reviewable code version himself in a subsequent round. We observed this procedure in GROMACS few times, and usually for the same reasons that reviewers swapped roles in ConQAT: Some changes are more time-consuming to explain than to realise, and are unlikely to cause objections from the original author. This because many code ideas and architecture decisions are still inherently difficult to explain [Bro87], even with the advent of design patterns [GHJV93]: Sometimes it is more efficient to let the reviewer, who had the idea for the change, do the rework. This is an idea of the more laissez-faire light-weight reviews, forbidden in formal review techniques like the Fagan inspection [Fag76].

Example 3 The reviewer performs a (non-trivial) change in the yellow code, and marks it green.



```
 * @author $Author: Tellkas $
 * @version $Rev: 26282 $
 * @levd.rating YELLOW Hash: DC8705768C67B4A26917DD4D97BB48BC
 */
@AConQATProcessor(description = "This processor reads .trx files that contain the test results of NU
public class TrxFileReader extends ConQATProcessorBase {

        /** {@ConQAT.Doc} */
        @AConQATKey(type = "String", description = "The duration of the test run")
        public static final String DURATION_KEY = "duration";

        /** {@ConQAT.Doc} */
        @AConQATKey(type = "String", description = "The id of the test case")
        public static final String ID_KEY = "id";
```

```
 * @author $Author: Tellkas $
 * @version $Rev: 26282 $
 * @levd.rating GREEN Hash: 80C2CD291E792BF5AF2C364A4344D58D
 */
@AConQATProcessor(description = "This processor reads .trx files that contain the test results
public class TrxFileReader extends ConQATProcessorBase {

        /** {@ConQAT.Doc} */
        @AConQATKey(type = "java.lang.String", description = "The duration of the test run")
        public static final String DURATION_KEY = "duration";

        /** {@ConQAT.Doc} */
        @AConQATKey(type = "java.lang.String", description = "The id of the test case")
        public static final String ID_KEY = "id";
```

In contrast to ConQAT, there must at least two (or more) reviewers in GROMACS.

Building the Database

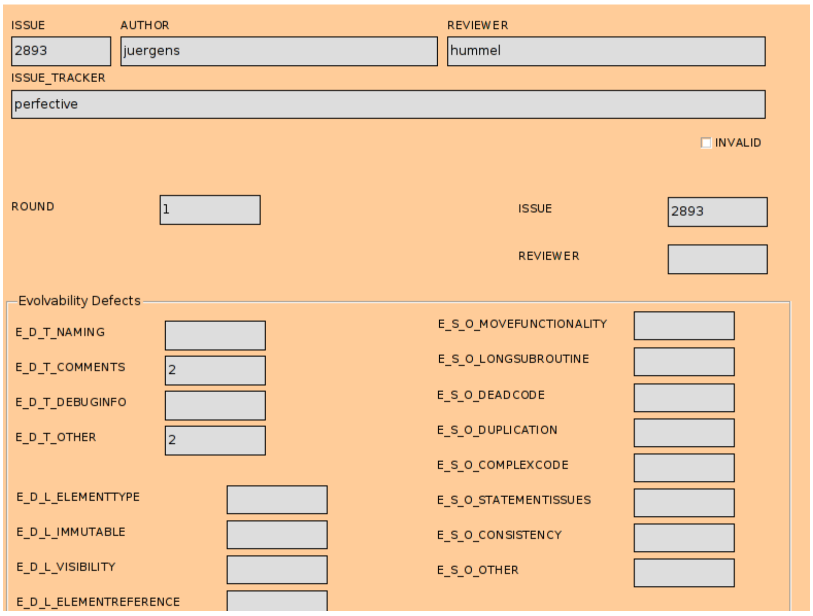
We collected our classfications of the findings with the help of a relational database. For the design of the database we used the Base component of the free office suite LibreOffice [Lib]. Figure 5.2 is an exemplary screenshot of our data input mask. We stored every data set (ConQAT Random, ConQAT 100, GROMACS) in its own database, but kept the structure of the tables identical across databases.

Study Procedure

Here we describe the technical details of how we carried out the study design.

Sampling of Issues

Since we sampled on a per issue basis, we needed back-references from the VCS to the CMS. In ConQAT and Gerrit, the commit message in the VCS references the issue ID from the CMS.



ISSUE	2893
AUTHOR	juergens
REVIEWER	hummel
ISSUE_TRACKER	perfective
☐ INVALID	
ROUND	1
ISSUE	2893
REVIEWER	

Evolvability Defects

E_D_T_NAMING		E_S_O_MOVEFUNCTIONALITY	
E_D_T_COMMENTS	2	E_S_O_LONGSUBROUTINE	
E_D_T_DEBUGINFO		E_S_O_DEADCODE	
E_D_T_OTHER	2	E_S_O_DUPLICATION	
		E_S_O_COMPLEXCODE	
E_D_L_ELEMENTTYPE		E_S_O_STATEMENTISSUES	
E_D_L_IMMUTABLE		E_S_O_CONSISTENCY	
E_D_L_VISIBILITY		E_S_O_OTHER	
E_D_L_ELEMENTREFERENCE			

Figure 5.2: Database input mask showing issue 2893. The mask is divided into two parts: The general per-issue information in the fields `ISSUE`, `REVIEWER`, `AUTHOR`, `ISSUE_TRACKER` and `INVALID`, and the per-review-round fields which represent the categories from appendix A.

We admitted only committers with a substantial amount of assigned issues into the sampling phase: Novices in the code have to adapt to the project first, which likely leads to bias in the distribution of the review categories in their issues: For example, we observed an increased number of findings and review rounds during their familiarisation phase with ConQAT. Therefore, we excluded all authors with fewer than ten assigned issues. Next, we excluded all issues that did not have an assigned reviewer in the CMS. Out of all the issues assigned to the remaining authors, we randomly picked ten issues as samples per author.

Sampling Tool To assist us in the sampling process for the data sets ConQAT Random and GROMACS, we developed a program for the automated randomly stratified sampling of issues. We developed a Java program that is able to read in data from the REST APIs of Redmine, Teamscale, and Gerrit. It gathers this data and unifies it in one coherent model. Based on our filtering preconditions and using Java's time-seeded random generator, the tool sampled the issues which we then manually assessed.

ConQAT Our observation period starts with the first issue in the CMS in 2005 and ends January 30th 2013 00:00, the last data point in our frozen SVN snapshot. An analysis of ConQAT showed that links between commits in the VCS and issues in the CMS have only been made since 2007. Our observation period is limited on the lower end by the introduction of a reference in the commit message to the issue. Some issues do not change code at all. Consequently, no code review is performed on these issues. Therefore, we excluded issues that do not have associated changed Java files in ConQAT. Furthermore, we are only interested in closed issues: If a review was performed on these issues, it must be finished by now. Under these constraints the number of suitable issues in ConQAT reduces from 3094 to 919, cf. figures 4.1 and 5.3.

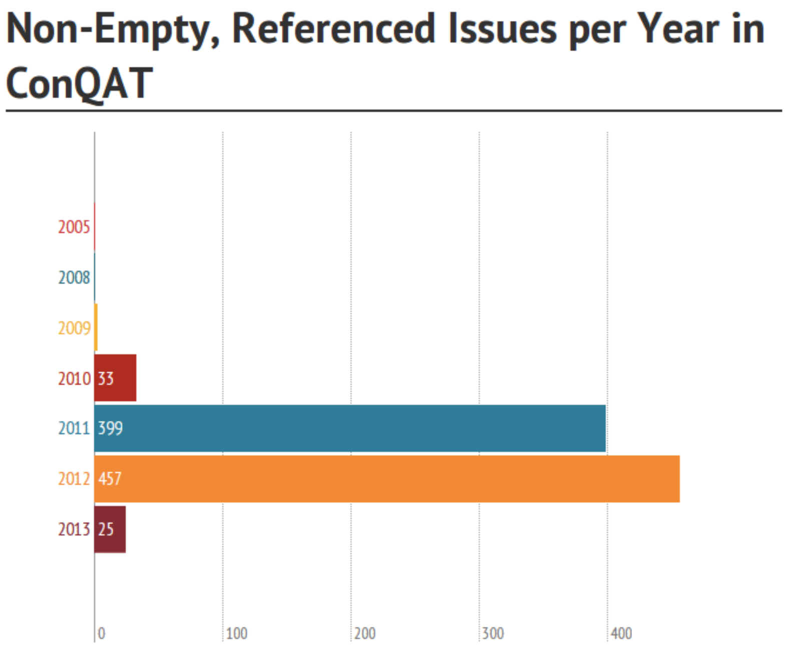


Figure 5.3: The number of issues created per year that have changed files associated with them. Data goes until 30th of January 2013. The total number of created issues is 919.

ConQAT has 13 authors fulfilling these preconditions, for which we sampled 130 different issues in ConQAT Random. For ConQAT 100, we looked at the most-recent one hundred issues after the filtering process.

GROMACS GROMACS developers started to use code reviews with the introduction of Gerrit on August the 3rd 2011. To compensate for an initial learning phase, our observation periods starts on November the 1st 2011. It ends on July the 1st 2013. Additionally, we only considered closed issues. This amounts to 293 issues in the observation period.

GROMACS has eight authors fulfilling our preconditions, for which we sampled 80 different issues.

In GROMACS, the review system Gerrit sits between the CMS and VCS. Particularly, for each issue that involves commits to the SVN a review ticket in Gerrit has to exist. However, one review ticket may reference several issues in the VCS. Therefore, two (or more) sampled issues may link to the same Gerrit ticket. In these cases, we assessed the Gerrit ticket only once for the first sampled issue, and for each other issue, referenced the first issue. This does not make the issues invalid, since a review was performed, but it sets the number of findings for the first issue to the accumulated number of all Gerrit tickets, and for the later issues to zero, which is arguably not accurate.

Assessing an Issue

In the classification process of the review changes we used three tools: Our own Eclipse Plugin, the Teamscale Web UI for evaluation of ConQAT, and the Gerrit Web UI for GROMACS.

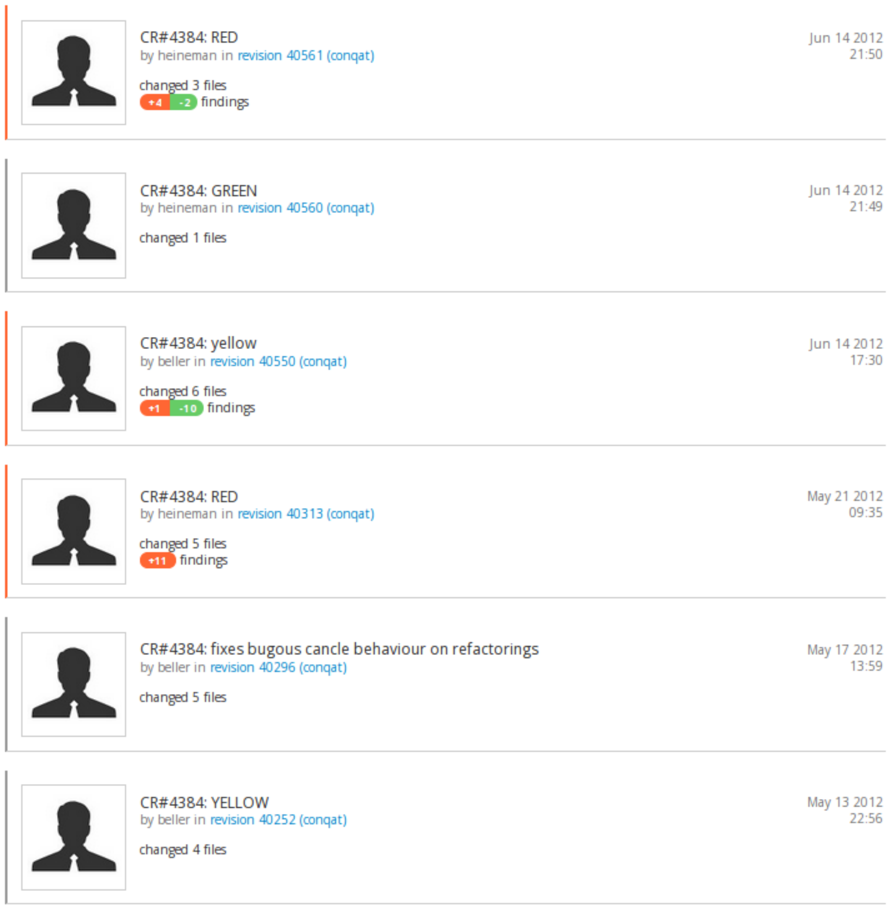
Teamscale Teamscale is “a quality analysis suite for continuous software quality control” [Tea]. At the time of writing this thesis, Teamscale was under development at CQSE GmbH: No stable version had yet been published. However, it allowed the analysis of ConQAT’s VCS repository, SVN, with a stringent history. Usually, if the commiter renames a file, this is handled as a delete and then an add operation in SVN. Even though the file contents may not differ, it is not possible to trace the origins of the newly added file to the old file. Teamscale provides mechanisms to follow the file’s history across such operations. A standard SVN log analysis would not have been sufficient, as ConQAT’s SVN includes many of these operations, leaving us with an incomplete history. If the review process of an issue stretches over long periods of time, it is likely to encounter “untraceable” SVN operations. Therefore, we configured a Teamscale instance with a repository mining of ConQAT’s source code. It gave us a continuous history of the project.

Teamscale provides a Web interface with basic support for source code and review comment assessment. It also provides a REST-ful web API, which we used as the data source for our tools.

Eclipse Plugin Our Eclipse Plugin, which integrated with Teamscale, allowed us to conveniently perform difference analyses on the ConQAT source code per review round. As figure 5.4 shows, both the “Perform Review” and the “Integrate Review” process (cf. figure 2.2) can comprise many commits. We are only interested in the change set at the end of each of the two processes, and not which changes occurred in-between the process (and might have been fixed by a later commit in the same sub-process). The commit-based diff offered by the Teamscale Web UI is often not sufficient if a sub-process consisted of more than one commit, nor suited for the efficient comparison of many files.

Our tool expects as input the issue number and two revision numbers corresponding to the start and end of a sub-phase of the review process. The plugin then requests all files touched by the specified issue for the given revisions from the Teamscale Server. It stores the files locally in the Eclipse workspace in their original tree structure. This enables us to use Eclipse’s Compare View to conveniently compare the two code versions from before the rework began to after the rework .

From the Teamscale Web UI we identify the revisions of the end of the review and rework processes, and compare each succeeding process artefact to the next. In formal reviews, only the reviewed and reviewable code of one round would need to be compared, but in ConQAT we need to monitor for changes from the reviewable to the reviewed code, cf. section 5.2 for an explanation.



CR#4384: RED
by heineman in revision 40561 (conqat)
changed 3 files
+4 -2 findings
Jun 14 2012
21:50

CR#4384: GREEN
by heineman in revision 40560 (conqat)
changed 1 files
Jun 14 2012
21:49

CR#4384: yellow
by beller in revision 40550 (conqat)
changed 6 files
+1 -10 findings
Jun 14 2012
17:30

CR#4384: RED
by heineman in revision 40313 (conqat)
changed 5 files
+11 findings
May 21 2012
09:35

CR#4384: fixes bugous cancle behaviour on refactorings
by beller in revision 40296 (conqat)
changed 5 files
May 17 2012
13:59

CR#4384: YELLOW
by beller in revision 40252 (conqat)
changed 4 files
May 13 2012
22:56

Figure 5.4: A succession of the first six commits for issue 4384 as displayed by the Teamscale Web UI. The first two commits by beller (rev. 40252, rev. 40296) belong to the original code writing process. The first review round ends with one review commit, rev. 40313. The integration of this review is done in one commit as well, rev. 40550. The next review process consists of two commits: rev. 40560 and 40561.

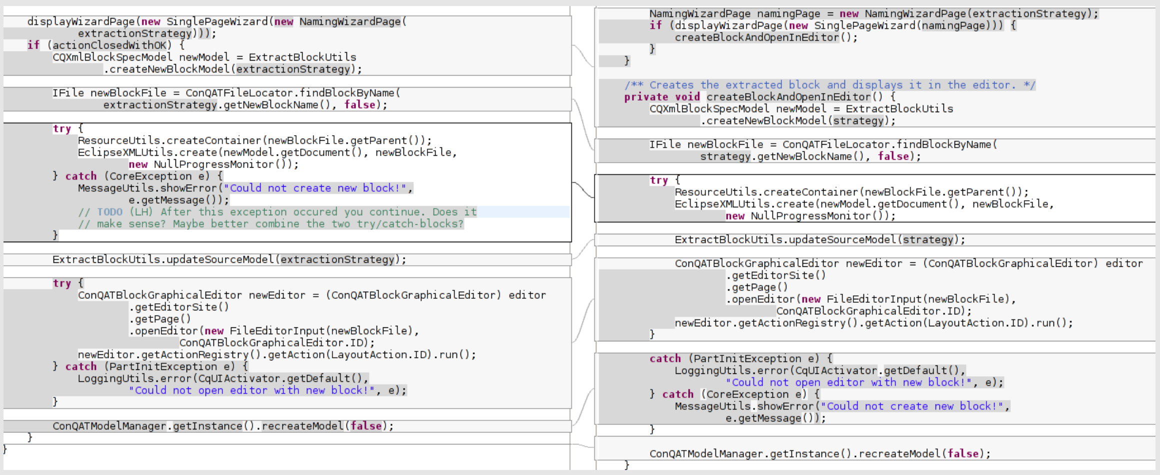
Example 4 For figure 5.4, we start with the comparison of rev. 40296—the original code—and 40313, which contains the review findings and changes of the first round. Code changes are classified as according to the defect types described in appendix A in the first review round. We then download rev. 40550, and compare this reviewable version with rev. 40313 to see how the author eliminated the detected problems. The second round's review ends with rev. 40561, which we compare to 40550 to see potential changes by the reviewer.

Classifying Changes Within an Issue

When we categorized the defects of an issue in the program code, we had the definitions of the defect categories and an overview graph as printed paper sheets in front of us. Many defects were quick to spot because they addressed and removed a finding noted by a reviewer.

However, difficulty arose when the changes were self-motivated, and involved large portions of code. It was often not evident which set of textual changes formed a logical, self-contained change unit with regard to the defect classification scheme: The scope of a change was not easy to determine.

Example 5 How many self-contained changes happened from left to right?

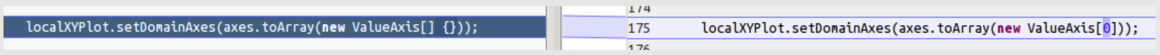


```
    displayWizardPage(new SinglePageWizard(new NamingWizardPage(
            extractionStrategy)));
    if (actionClosedWithOK) {
        CQXmlBlockSpecModel newModel = ExtractBlockUtils
                .createNewBlockModel(extractionStrategy);

        IFile newBlockFile = ConQATFileLocator.findBlockByName(
                extractionStrategy.getNewBlockName(), false);

        try {
            ResourceUtils.createContainer(newBlockFile.getParent());
            EclipseXMLUtils.create(newModel.getDocument(), newBlockFile,
                    new NullProgressMonitor());
        } catch (CoreException e) {
            MessageUtils.showError("Could not create new block!",
                    e.getMessage());
            // TODO (LH) After this exception occured you continue. Does it
            // make sense? Maybe better combine the two try/catch-blocks?
        }

        ExtractBlockUtils.updateSourceModel(extractionStrategy);

        try {
            ConQATBlockGraphicalEditor newEditor = (ConQATBlockGraphicalEditor) editor
                    .getEditorSite()
                    .getPage()
                    .openEditor(new FileEditorInput(newBlockFile),
                            ConQATBlockGraphicalEditor.ID);
            newEditor.getActionRegistry().getAction(LayoutAction.ID).run();
        } catch (PartInitException e) {
            LoggingUtils.error(CqUIActivator.getDefault(),
                    "Could not open editor with new block!", e);
        }

        ConQATModelManager.getInstance().recreateModel(false);
    }
}
```

```
        NamingWizardPage namingPage = new NamingWizardPage(extractionStrategy);
        if (displayWizardPage(new SinglePageWizard(namingPage))) {
            createBlockAndOpenInEditor();
        }
    }

    /** Creates the extracted block and displays it in the editor. */
    private void createBlockAndOpenInEditor() {
        CQXmlBlockSpecModel newModel = ExtractBlockUtils
                .createNewBlockModel(strategy);

        IFile newBlockFile = ConQATFileLocator.findBlockByName(
                strategy.getNewBlockName(), false);

        try {
            ResourceUtils.createContainer(newBlockFile.getParent());
            EclipseXMLUtils.create(newModel.getDocument(), newBlockFile,
                    new NullProgressMonitor());

            ExtractBlockUtils.updateSourceModel(strategy);

            ConQATBlockGraphicalEditor newEditor = (ConQATBlockGraphicalEditor) editor
                    .getEditorSite()
                    .getPage()
                    .openEditor(new FileEditorInput(newBlockFile),
                            ConQATBlockGraphicalEditor.ID);
            newEditor.getActionRegistry().getAction(LayoutAction.ID).run();
        }

        catch (PartInitException e) {
            LoggingUtils.error(CqUIActivator.getDefault(),
                    "Could not open editor with new block!", e);
        } catch (CoreException e) {
            MessageUtils.showError("Could not create new block!",
                    e.getMessage());
        }

        ConQATModelManager.getInstance().recreateModel(false);
    }
```

Furthermore, we found it difficult to infer from only the comparison of two source code versions which category an undocumented change belonged to: In rare cases, it was difficult to assess whether the change had functional implications, or not.

Example 6 Although the scope of this change is easy to determine, it is difficult to rate the defect as functional or non-functional without a deep knowledge of Java and the underlying system.



```
localXYPlot.setDomainAxes(axes.toArray(new ValueAxis[] {}));
```

```
localXYPlot.setDomainAxes(axes.toArray(new ValueAxis[0]));
```

One code change is rated in precisely one category. If we thought more than one defect category for one change suitable, we used the most precise fitting, which explained best why a change was conducted.

Example 7 If a variable's name `resultGood` is fine for itself, but all other variable names in the class begin with an adjective—such as `badResult`—two categorizations for the change from `resultGood` to `goodResult` are thinkable: A Naming Defect, or a Consistency Defect. In these cases, we opted for the Consistency Defect because the rename operation was performed out of consistency reasons, and not because the original name was bad per se.

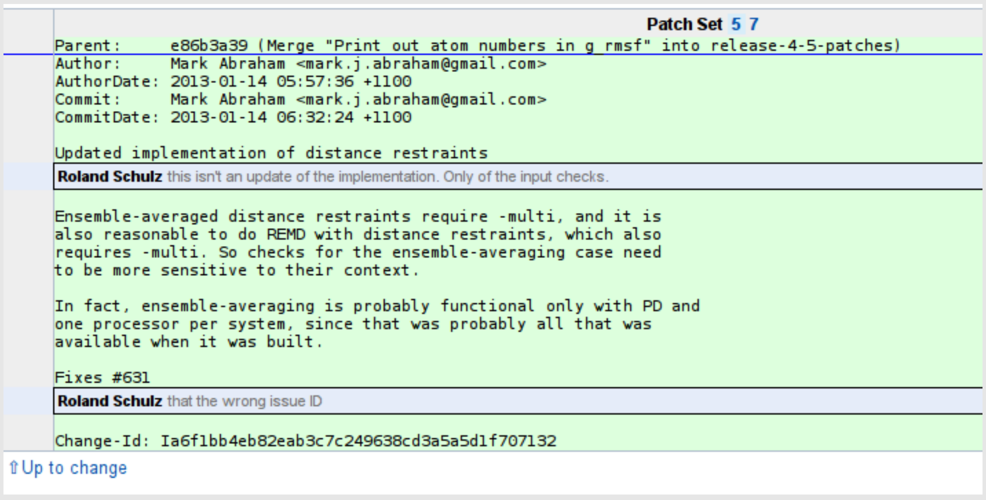
While we tried to rate changes as fine-granular and precise as possible, we preferred to rate larger changes with a recognizable functional change in the program as one larger functional defect. If we could rate a defect as either evolvable or functional, we preferred the functional category: In our understanding, the effects of a functional change in the program outweigh evolvability issues. [ML09] argues similarly: "If the researcher was not sure and it was not possible to ask the author of the code, a functional defect class was chosen."

Idiosyncrasies of ConQAT and GROMACS

Two subtle peculiarities are the result of different reviewing processes in ConQAT and GROMACS that hinder the comparison of the two. In this section, we explain their nature, and how we resolved them to make ConQAT and GROMACS as comparable as possible.

Gerrit allows to review and alter all parts of a commit, which is not possible in LeVD style reviews, since the review is not performed on a commit, but on a file basis. This allows Gerrit users to find a complete new findings category `E_META`, which cannot be detected and corrected with LeVD. Examples for defects in this category are typos in the commit message and more substantially the addition and the correction of referenced issues. This leads to a better traceability between the CMS and VCS, which increases maintainability of the project. Since we do not posses such findings for ConQAT, we left the `E_META` category out in the comparison.

Example 8 GROMACS review of a commit message, showing a `E_META` defect.



```
Parent:     e86b3a39 (Merge "Print out atom numbers in g_rmsf" into release-4-5-patches)
Author:     Mark Abraham <mark.j.abraham@gmail.com>
AuthorDate: 2013-01-14 05:57:36 +1100
Commit:     Mark Abraham <mark.j.abraham@gmail.com>
CommitDate: 2013-01-14 06:32:24 +1100

Updated implementation of distance restraints
```

Roland Schulz this isn't an update of the implementation. Only of the input checks.

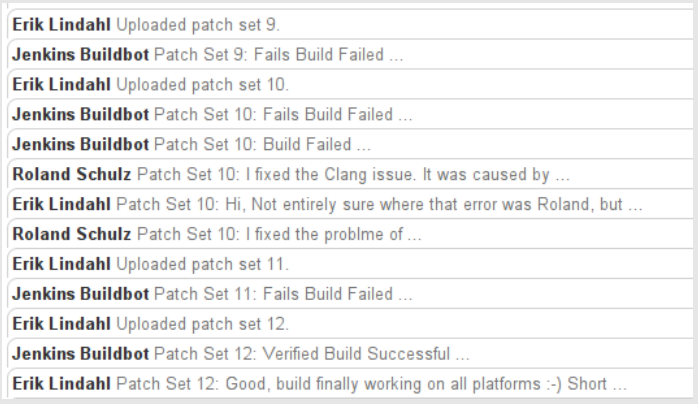
```
Ensemble-averaged distance restraints require -multi, and it is
also reasonable to do REMD with distance restraints, which also
requires -multi. So checks for the ensemble-averaging case need
to be more sensitive to their context.

In fact, ensemble-averaging is probably functional only with PD and
one processor per system, since that was probably all that was
available when it was built.

Fixes #631
```

Roland Schulz that the wrong issue ID

```
Change-Id: Ia6f1bb4eb82eab3c7c249638cd3a5a5d1f707132
```

`F_BUILD` denotes build failures detected by the automated Jenkins build job in Gerrit. Such failures do not show up in ConQAT because it uses a mailinglist-based blame system for reporting broken builds. As a reaction to a blame mail the original author usually issues a fixing commit within hours of his breaking changes. The time it takes him to fix the build is typically much shorter than the time until the review starts. Therefore, in ConQAT, build fixes will normally go unnoticed and do not show up as an extra review round, as they do in Gerrit: The number of review rounds in GROMACS is potentially higher, with a smaller findings count—for each `F_BUILD` defect, Gerrit automatically creates a new review with the pseudo-reviewer "Jenkins" with only one defect in it. To make GROMACS's classification scheme compatible with ConQAT's we left out the `F_BUILD` catgeory in the further analyses of our case study. Since we are assessing the benefits of manual code reviews in this thesis, the number of automated building failure findings is not relevant.

Example 9 The automated Jenkins build integration in Gerrit recognizes a broken build after uploading a patch set and warns the author Erik Lindahl of this. We can see four review rounds and three `F_BUILD` defects in this example.


Erik Lindahl Uploaded patch set 9.
Jenkins Buildbot Patch Set 9: Fails Build Failed ...
Erik Lindahl Uploaded patch set 10.
Jenkins Buildbot Patch Set 10: Fails Build Failed ...
Jenkins Buildbot Patch Set 10: Build Failed ...
Roland Schulz Patch Set 10: I fixed the Clang issue. It was caused by ...
Erik Lindahl Patch Set 10: Hi, Not entirely sure where that error was Roland, but ...
Roland Schulz Patch Set 10: I fixed the problme of ...
Erik Lindahl Uploaded patch set 11.
Jenkins Buildbot Patch Set 11: Fails Build Failed ...
Erik Lindahl Uploaded patch set 12.
Jenkins Buildbot Patch Set 12: Verified Build Successful ...
Erik Lindahl Patch Set 12: Good, build finally working on all platforms :-) Short ...

Results and Implications

Figures 5.5 to 5.7 show the number of absolute changes per category for our evaluation of ConQAT Random, ConQAT 100 and GROMACS. The graphs show on the x axis abbreviated names of the categories from appendix A. The sub-categories of the top-level category evolvability are printed in shades of blue, and the functional defects in orange. On the y axis the absolute number of defects found in each category is plotted. In the following, we interpret the results from these graphs.

Invalid Issues

We could not include all of the sampled issues in this study: Some did not undergo the complete review process—for example, the review was abandoned in the process—, the review was done as part of another issue, which we did not sample, the issue contained large portions of code changes in closed-source repositories, or the issue was so complicated with so many committers that we couldn't fully comprehend the proceedings. Additionally, some reviews were not fully performed within our observation period, but we sampled them nevertheless, since we could not a-priori safely determine the date an issue

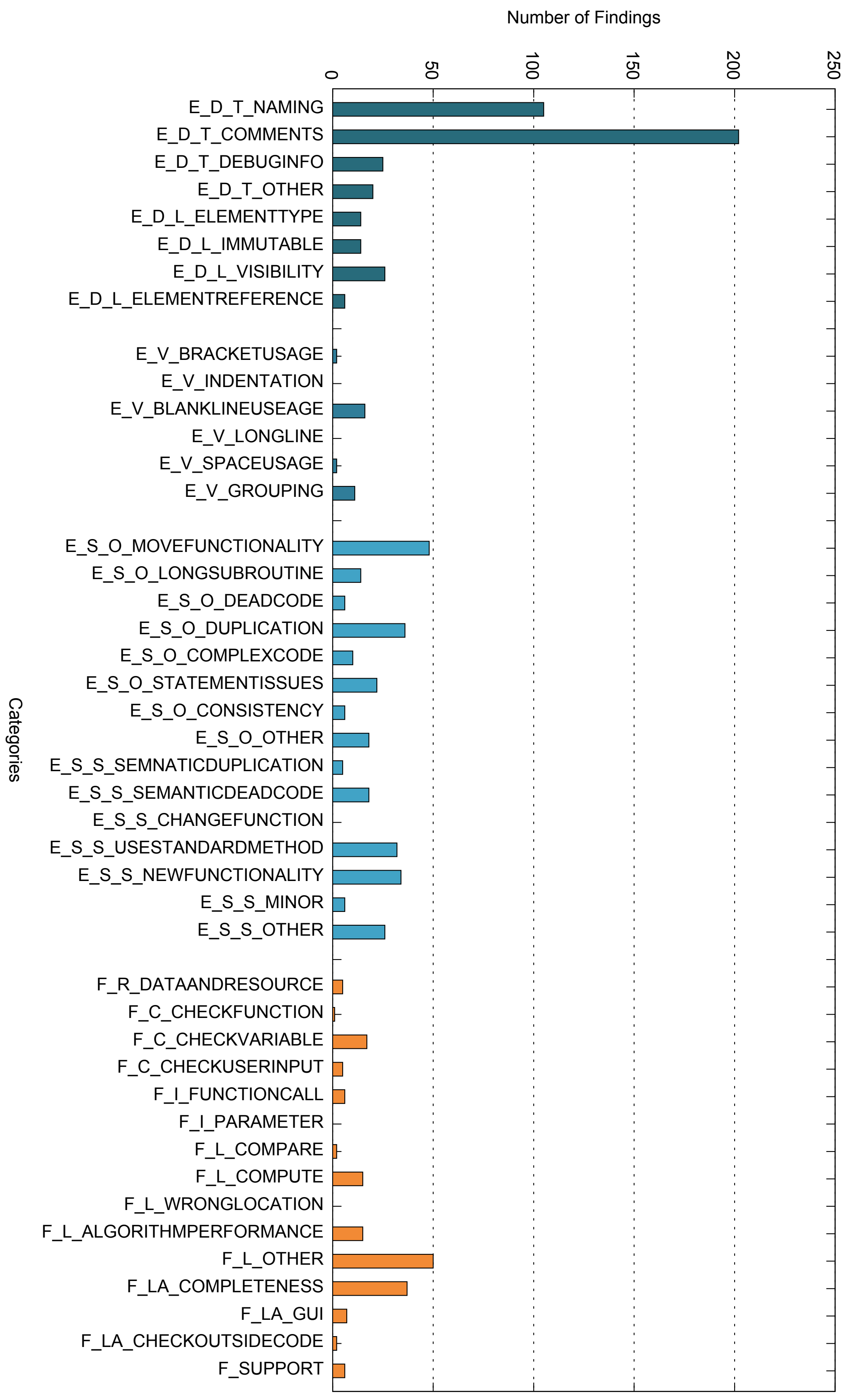


Figure 5.5: The defect distribution profile (number of absolute findings in each category) for 100 randomly sampled ConQAT issues. Total number of defects: 892.

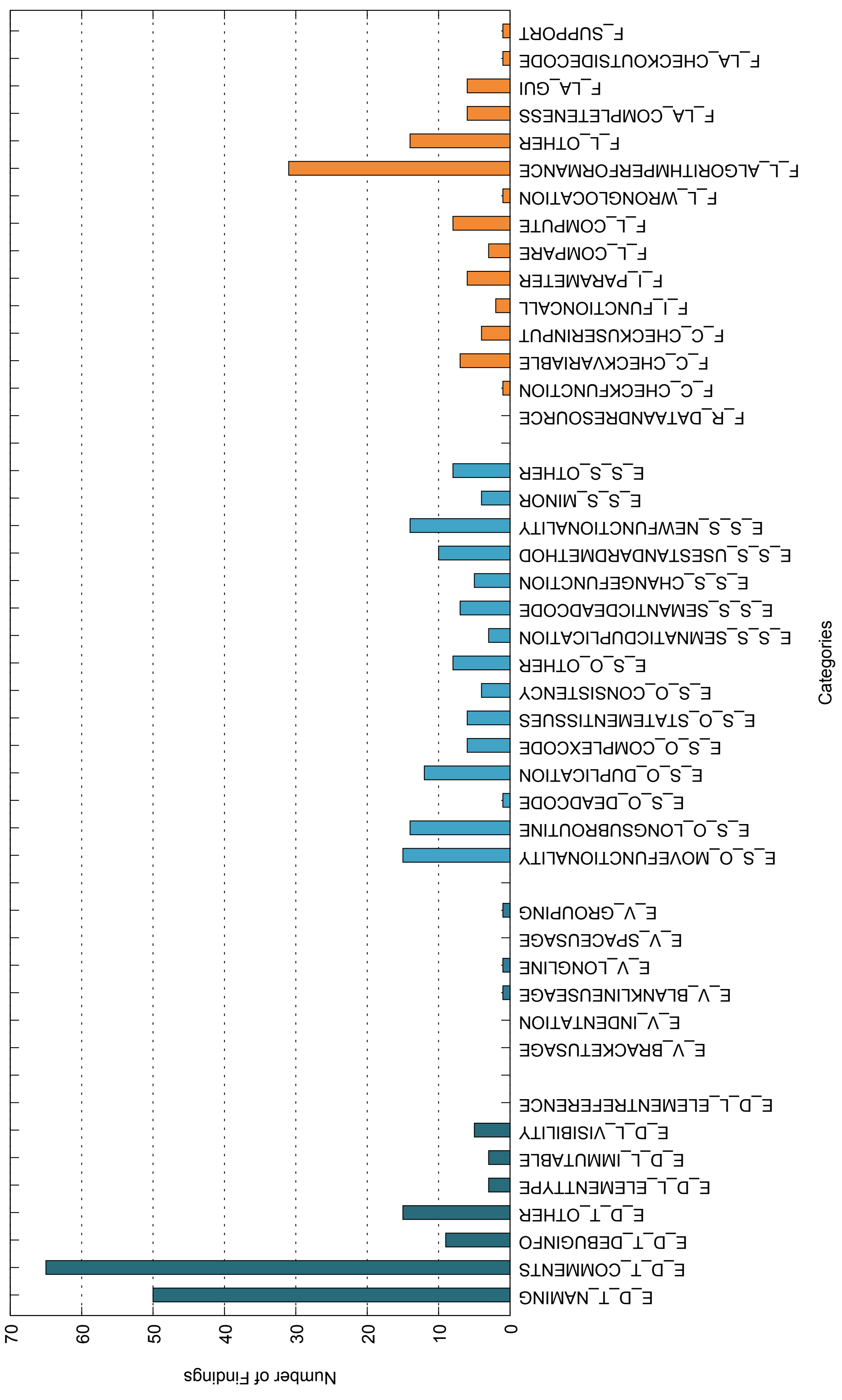


Figure 5.6: The defect distribution profile (number of absolute findings in each category) for the most recent 100 ConQAT issues. Total number of defects: 361.

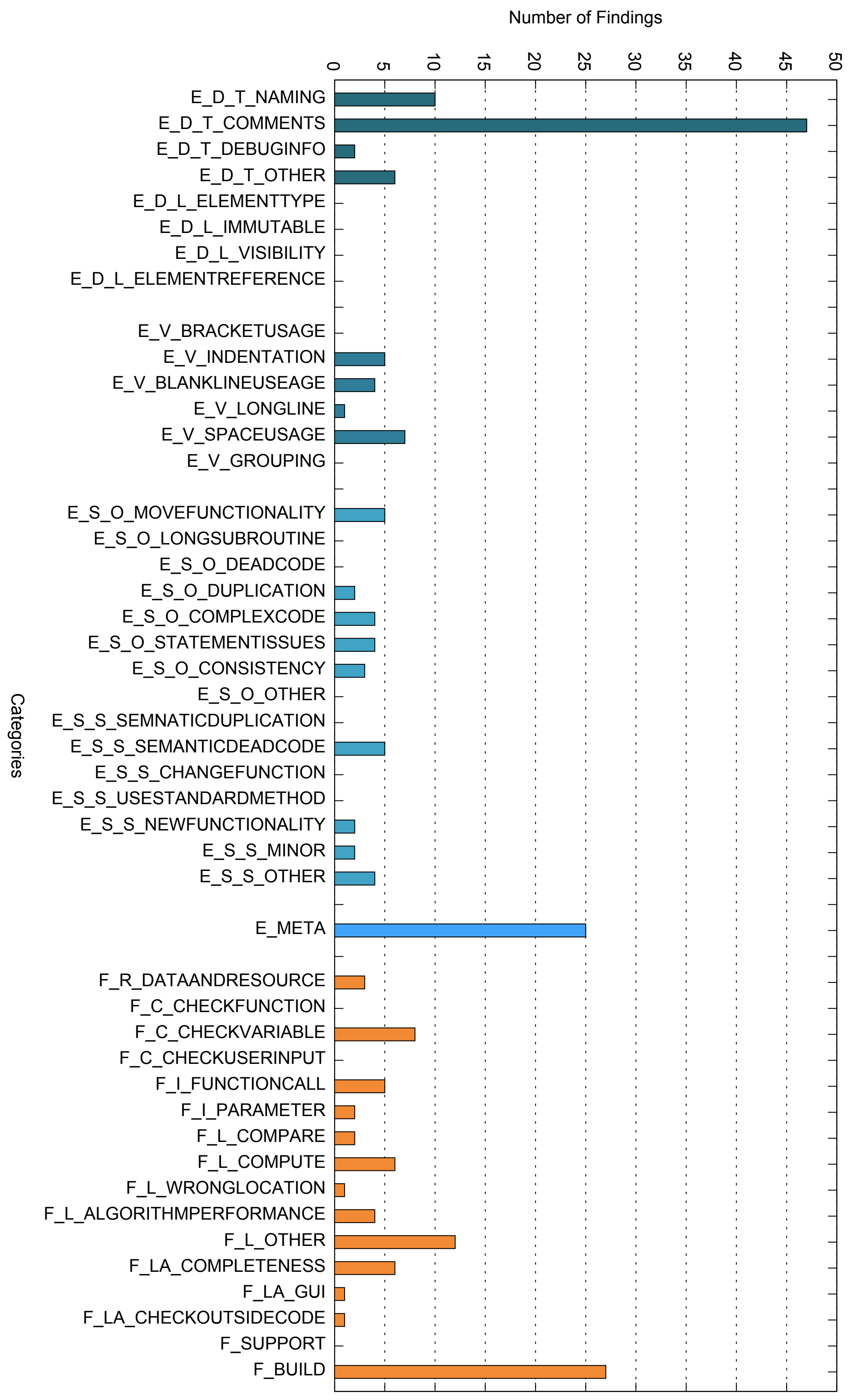


Figure 5.7: The defect distribution profile (number of absolute findings in each category) for the sampled issues in GROMACS. Total number of defects (without `F_BUILD` and `E_META`): 216 (164).

had been closed. Since we had a sufficiently large sample at hand, we did not include such dubious issues in our case study.

ConQAT Random had 100 valid issues out of 128 issues in total (78.1%). ConQAT 100 had 89 valid issues out of 100 issues in total (89%). GROMACS had 60 valid issues out of 80 issues in total (75.0%). The percentage of valid issues is similar across systems, so we do not assume a biased preselection of the sampled issues.

In order to avoid bias on a per-author level, we took care that the number of invalid issues per author was not higher than three, so as to not distort the final results because of fewer analysed defects from a certain author. This was only the case for one author in ConQAT Random (who had seven invalid issues), for whom we re-sampled issues.

Number of Defects

Our first distinctive observation is the number of absolute defects per sample. Although sample sizes are roughly comparable ($|\text{ConQAT 100}|$ is $0.91 \times |\text{ConQAT Random}|$, and $|\text{GROMACS}|$ is $0.63 \times |\text{ConQAT Random}|$), there were absolutely fewer defects in both ConQAT 100 and GROMACS: Based on the number of findings from ConQAT Random, we would expect to find around 810 defects in ConQAT 100, whereas we found only 361 (44% of the expected value). In GROMACS we would expect 558 defects, but found only 164 (29% of the expected value). Our intuition during the manual assessment of the GROMACS reviews is in alignment with this observation: Even though more reviewers are involved in GROMACS, the attention to detail seemed much lower compared to ConQAT.

A related distinctive feature is the deviating number of defects per review. Figure 5.8 illustrates this observation: ConQAT Random has a range from 0 to 208 defects per issue maximally. Its median is 2 defects per issues, its average 8.81. 75% of issues have between 0 and 6 defects. ConQAT 100 has a range from 0 to 110 defects per issue maximally. Its median is 0 defects per issue, its average 4.00. 75% of issues have between 0 and 2 defects. GROMACS has a range from 0 to 93 defects per issue maximally. Its median is 0 defects per issue, its average 3.24. 75% of issues have between 0 and 2 defects.

Some issues have extreme outliers, their defect count being orders of magnitude higher than the reported median or average for each system. An explanation could be that most issues in the CMS are relatively small and well-split up. However, sometimes a really large change request with lots of work arises. The possibility that in both ConQAT and GROMACS the review of one issues is sometimes performed in the scope of another issue could also contribute to these high values: The highest outliers for both systems contained references from several issues.

Defect Types

Across all systems, the defect category with the highest occurrence rate is `E_D_T_COMMENTS`. Recent research has found that comments in the code are often trivial, difficult to understand, or outdated [SHJ13]. Our results show that reviews lead to revised comments, which indicates that reviews could be a mechanism to counter problems associated with or caused by comments.

The second prominent defect category are `E_D_T_NAMING` defects. In the ConQAT samples, there are 25% to 50% fewer `NAMING` than `COMMENTS` defects, while this is still by far

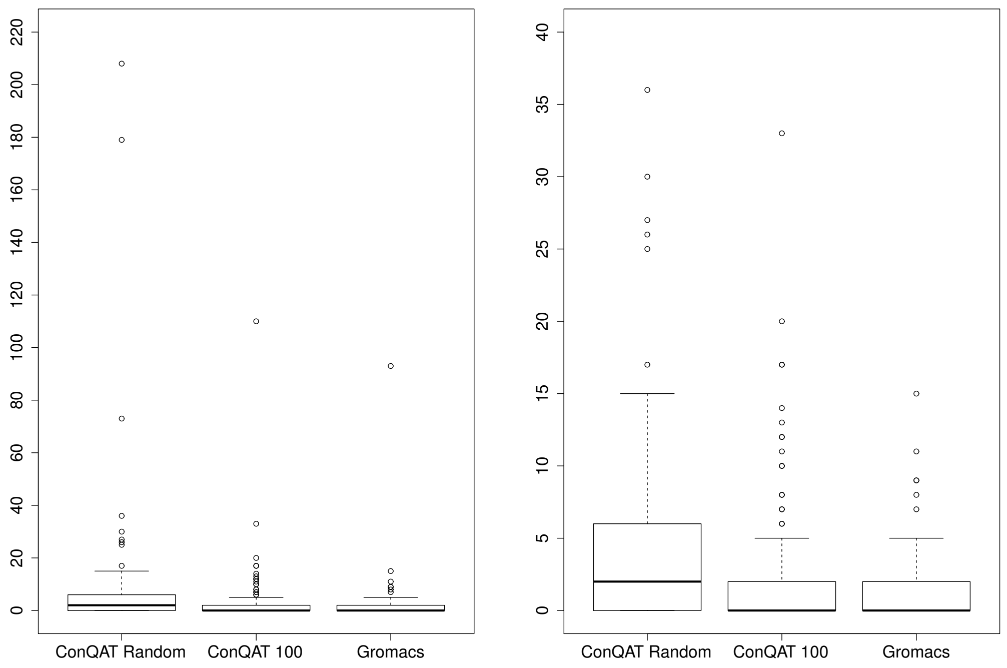


Figure 5.8: Box-and-whisker plots for the number of defects found per issue in the three samples. The plot on the right is a zoomed-in version of the left-hand side plot to better illustrate the distribution in the range between 0 and 40 defects per issue.

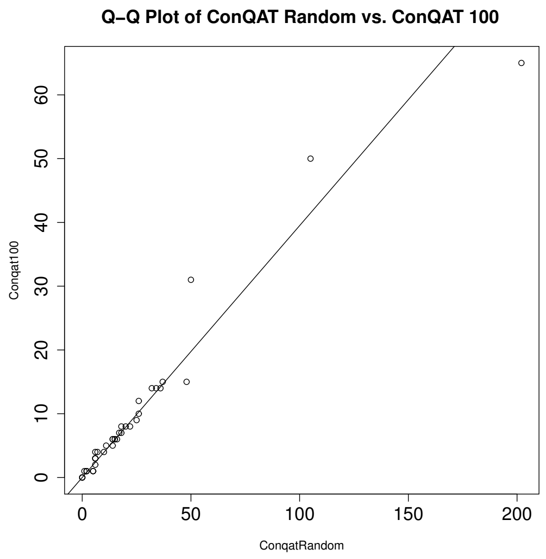
the second highest value for any finding category. In GROMACS, `NAMING` defects account for far fewer defects than `COMMENTS` defects, rouhgly 80%, and they are only the third largest category by a small margin to the `F_CHECKVARIABLE` defect.

In GROMACS no defects from the `E_D_L_*` sub-category were fixed. We can explain this with the fact that GROMACS is a C system, and C does not support these object orientation concepts. Furthermore, no `E_V_BRACKETUSAGE` defect was discovered. This could be indicative of two circumstances: Either all GROMACS developers use brackets consistently, or the review guidelines do not mandate a consistent bracket usage. ConQAT style guidelines require the use of curly brackets even in one liners where they would be syntactically redundant. Consequently, reviewers found some violation of this rule. ConQAT on the other hand has very few `E_V_*` defects because the automatic code formatter takes care of most of those.

A larger portion of defects is solved in the `E_S_ORGANIZATION` and `E_S_SOLUTION` subcategories in both ConQAT samples than in GROMACS. Defects in this category typically require an indepth examination of the reviewable code, as it is not trivial for a reviewer to detect when code is dead or duplicated, or when a standard method could be used instead. Together with the observation that GROMACS does have a similar amount of trivial changes like `E_D_T_NAMING`, we could reason that ConQAT has more in-depth reviews than GROMACS. This holds under the assumption that the quality of the original code in

GROMACS is similar to ConQAT—and we have no indication to assume otherwise.

Similarity of Review Distributions



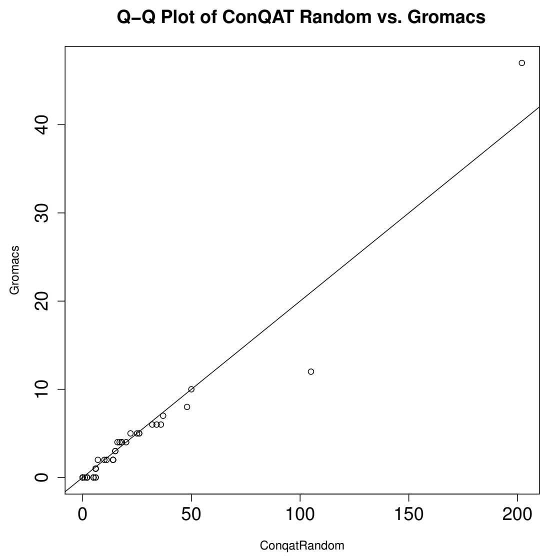


Figure 5.9: Q-Q Plots for the number of defects per category show the relative similarity of the defect distributions.

We have already established that the defect distributions for our three samples is similar by "overlaying" the relative distribution profile of the three samples. However, this is only a rough estimator of how close the distribution are.

To answer the question precisely, we plot Q-Q diagrams of Conqat 100 versus Conqat Random and Gromacs versus Conqat Random in figure 5.9. Essentially, the nearer the data points lie to the inscribed diagonal, the better the fit between the two distributions compared in the diagram. The theory of Q-Q diagrams is further explained in [WG68]. As we can see, both distributions are very similar to ConQAT Random. A comparison between the normal distribution and ConQAT Random shows a significantly greater offset, cf. figure 5.10. Therefore, our manual observation from prior chapters seems justified: The detailed defect distributions between the three samples is very similar.

5.3 Distribution Between Maintenance and Functional Defects

Siy and Votta report a 60:20 distribution of evolvability to functional defects [SV01]. Mäntylä and Lassenius confirm this ratio for two other projects, reporting distributions of 71:21 and 77:13 [ML09]. However, El Emam and Wieczorek [EW98] and R. Chillarege et al. [CBC$^+$92] report contradicting distributions, stating a ∼50:50 distribution for two systems and a ∼20:80 ratio for one system, respectively. Thus, further research on the distribution of evolvability and functional defects is needed.

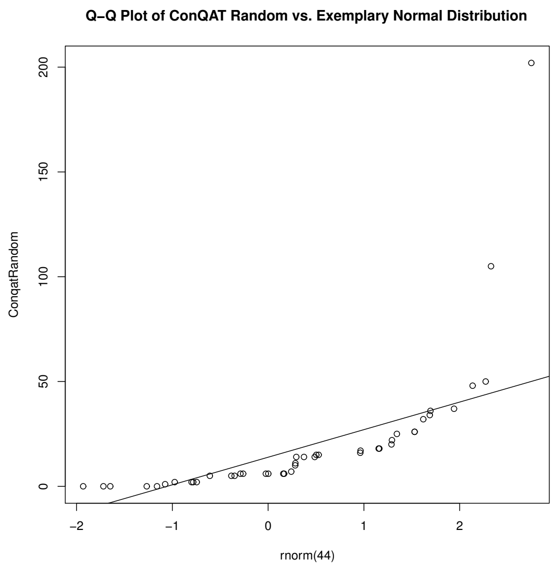


Figure 5.10: Q-Q Plots for the number of defects per category show the relative dissimilarity to an exemplary normal distribution of defect counts.

RQ 2 *What is the distribution between evolvability and functional defects in the OSS systems from RQ 1?*

This study is a replication of the first study performed in [ML09], and therefore confirmatory in its nature. To answer the research question, we use the data set generated for RQ 1 and classify the sampled fine-granular defects into the two top-level groups: Evolvability and functional defects. Since we re-used the dataset,the study design and procedure from section 5.2 apply to RQ 3 as well: Most important, in order to be able to compare it with ConQAT, we left out the `E_META` and `F_BUILD` categories from the GROMACS dataset.

Results and Implications

Figure 5.11 presents the ratio of evolvability and functional defects in our three data sets, and puts it in context with the values reported by [ML09]. Since we do not have a false positive category, we took the ratios from [ML09] sans false positives. As the graph shows, the resulting ratios among the four systems are relatively near each other, within a range of ten percentage points.

Figure 5.12 illustrates the difference in distributions when we include `E_META` and `F_BUILD` defects in GROMACS: Because build failures are frequent, the distribution is displaced in favour of functional defects. We would expect something similar for ConQAT, could we count build failures. However, we reason that these automatic findings are irrelevant for the quantification of manual code reviews.

In ConQAT Random we found more evolvability defects than in all other samples, 5 percentage points above 75%. ConQAT 100 hits the 75:25 ratio almost exactly. GROMACS has a slightly lower amount of evolvability defects at 68.9%. The uniformity of the result

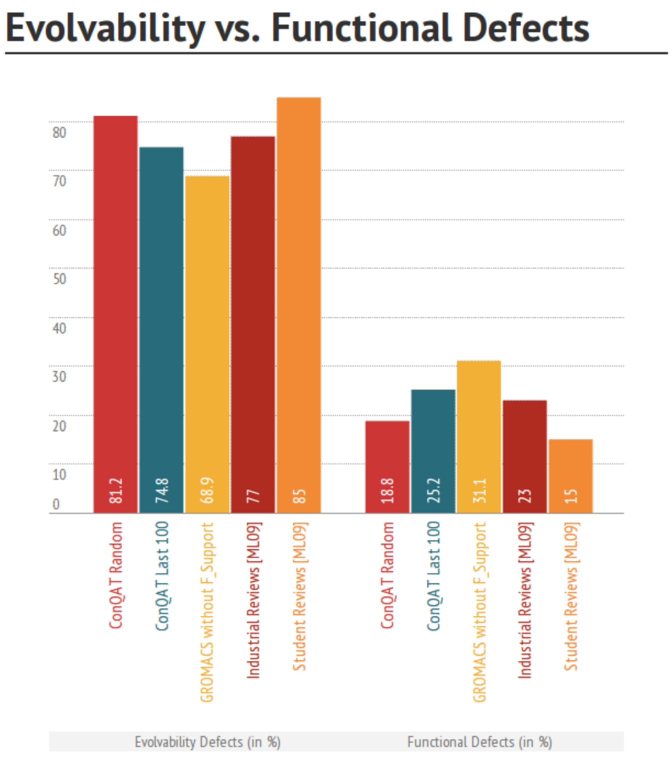


Figure 5.11: The ratio of evolvability and functional defects plotted against each other in the three samples from our case study, and the two samples from [ML09] (excluding false positives).

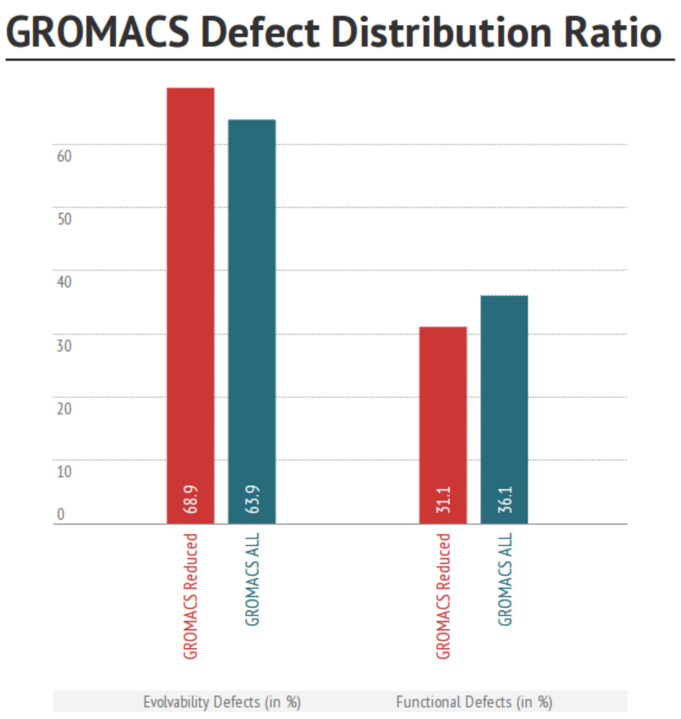


Figure 5.12: A comparison of the distributions of the evolvability and functional defects in GROMACS without and with `E_META` and `F_BUILD` defects.

is particularly interesting, as ConQAT and GROMACS are written in a different programming languages and development models by other people with strongly diverging review processes.

5.4 Usage of Code Review Findings

The effectiveness of reviews is often debated [SV01, WRBM97b, KP09b]. However, besides cost models, there has been little research on how many of the review findings lead to changes in the system, and how many are disregarded. Additionally, changes might be made by the author based on no particular review comment. If many or most of the review findings are discarded, we could assume that reviews are an inefficient approach to quality control.

RQ 3 *What is the ratio between accepted, self-motivated and disregarded review changes in the systems from RQ 1?*

This study is exploratory in nature. Since we re-used the dataset, the study design and procedure from section 5.2 apply to RQ 3 as well.

Study Procedure

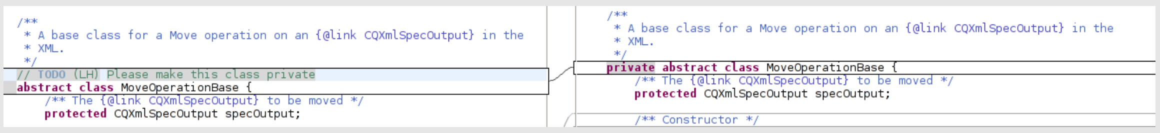
To answer this research question, we use the data set generated for RQ 1 and classified each change from RQ 1 as either triggered by a review comment, self-motivated or discarded (cf. chapter 2 for a detailed explanation on the motivations for a change).

In our study we saved how many findings of which type happened, for each review round individually. We do no treat individual changes as a database entry of their own. This would have allowed us to say exactly which types of findings were discarded and which were self-motivated. However, we did not notice abnormalities in the distribution of discarded or self-motivated changes during our manual assessment of the findings (e.g. we did not notice unproportionally many self-motivated changes were naming defects or similar). As we do not expect deviations from a proportionate distribution, we left out the time consuming tracking and modelling of individual changes in the data acquisition phase.

In contrast to RQ 1, where we had sometimes difficulties to find the correct defect category for a change, we could determine the motivation for a change easily most times. This was because the majority of changes was triggered by a review comment. Example 10 shows a typical review-triggered change. If the author didn't like a suggestion, he usually contradicted as a follow-up, making the determination of the "agreed discarded" group easy in most cases. Example 11 demonstrates this in the source code. Self-motivated changes had the same problem as RQ 1 with regard to scope, but once changes were identified, their motivation was relatively obvious. Examples 2, 3 and 6 show self-motivated changes from the reviewer.

Apart from these content-related reasons, we only had three categories to choose from, which makes it easier to find the correct category. Additionally, the three categories are orthogonal with regard to their definition.

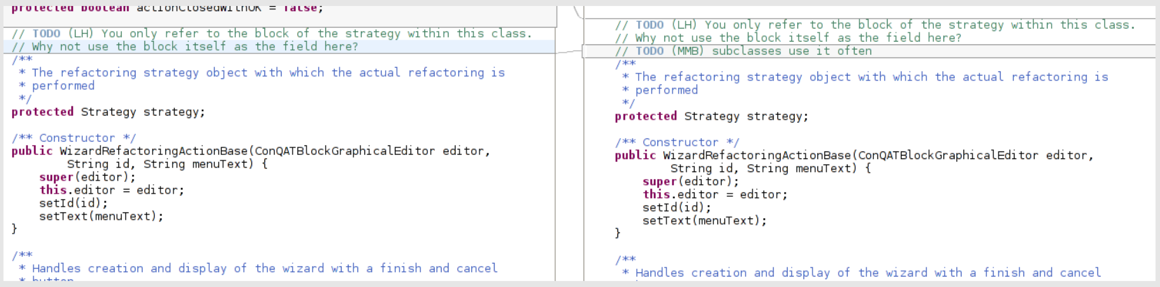
Example 10 `E_D_VISIBILITY` defect triggered by a review comment.



```
/**
 * A base class for a Move operation on an {@link CQXmlSpecOutput} in the
 * XML.
 */
// TODO (LH) Please make this class private
abstract class MoveOperationBase {
    /** The {@link CQXmlSpecOutput} to be moved */
    protected CQXmlSpecOutput specOutput;
```

```
/**
 * A base class for a Move operation on an {@link CQXmlSpecOutput} in the
 * XML.
 */
private abstract class MoveOperationBase {
    /** The {@link CQXmlSpecOutput} to be moved */
    protected CQXmlSpecOutput specOutput;

    /** Constructor */
```

Example 11 The reviewer proposes an `E_STRUCTURE_SOLUTION_OTHER` change. The author does not agree and shortly explains his reasons. In the next round, the reviewer accepted to leave the file as-is, and therefore the change was discarded in unison.



```
protected boolean actionClosedWithOK = false;

// TODO (LH) You only refer to the block of the strategy within this class.
// Why not use the block itself as the field here?
/**
 * The refactoring strategy object with which the actual refactoring is
 * performed
 */
protected Strategy strategy;

/** Constructor */
public WizardRefactoringActionBase(ConQATBlockGraphicalEditor editor,
        String id, String menuText) {
    super(editor);
    this.editor = editor;
    setId(id);
    setText(menuText);
}

/**
 * Handles creation and display of the wizard with a finish and cancel
 * button
```

```
    // TODO (LH) You only refer to the block of the strategy within this class.
    // Why not use the block itself as the field here?
    // TODO (MMB) subclasses use it often
    /**
     * The refactoring strategy object with which the actual refactoring is
     * performed
     */
    protected Strategy strategy;

    /** Constructor */
    public WizardRefactoringActionBase(ConQATBlockGraphicalEditor editor,
            String id, String menuText) {
        super(editor);
        this.editor = editor;
        setId(id);
        setText(menuText);
    }

    /**
     * Handles creation and display of the wizard with a finish and cancel
```

Results and Implications

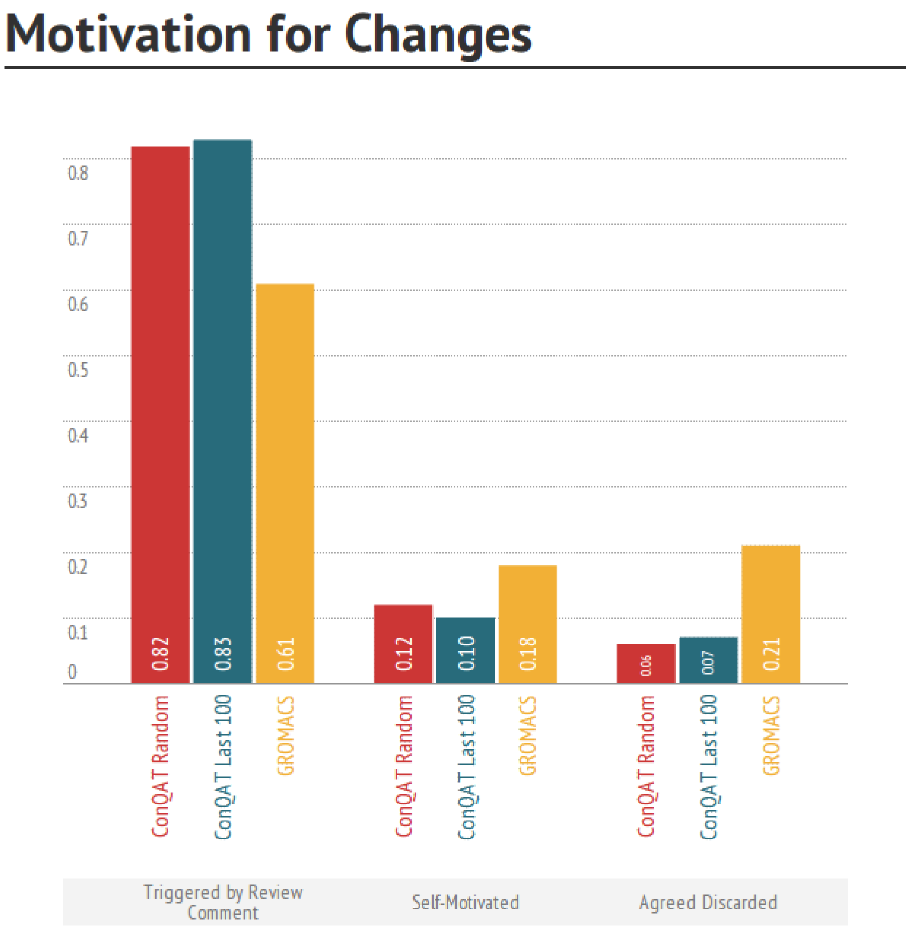


Figure 5.13: The motivation for changes in the ConQAT, ConQAT 100 and GROMACS.

Figure 5.13 depicts the results of this study. All three samples show two uniting features: The changes triggered by a review comment form the main group of recognized defects. Of the number of actual changes in the system, they make up 87% of changes for ConQAT Random, 89% for ConQAT 100 and 77% for GROMACS. The percentage of self-motivated or discarded changes is considerably lower.

The number of realised changes can be modelled as the sum of triggered and self-motivated changes. Thus, we have 94% realised changes for ConQAT Random, 93% for ConQAT 100 and 79% for GROMACS. The percentage of rejected changes was 6 to 7% for ConQAT systems, and 21% for GROMACS.

We can conclude from these numbers, that a majority of review suggestions is realised. Therefore, reviews appear to make sense. Yet, self-motivated changes remain an important part of changes in light-weight reviews.

5.5 Threats to Validity

We describe factors that threaten the validity of our case study on defect types, the basis for RQ 1 to 3, and show how we mitigated them.

Internal Threats

Internal threats are factors that could affect our measurements, but which we did not control for. There are several internal factors that could threaten our results, most of which we could mitigate.

1. Hawthorne Effect
   The Hawthorne Effect refers to the phenomenon that participants of case studies perform above average because they know they are being watched [Ada84]. We could rule out this effect because we started our studies posteriori: Neither the authors nor reviewers from ConQAT or GROMACS knew we would later undertake this study when they made their contributions.

2. Biased Sampling
   Thanks to stratified randomized sampling, we captured a representative sample of ten issues per regular author. This way, no single author has an over-proportional impact on the result. Particularly, we exclude issues from authors that had only minor influences on the systems. In ConQAT, for example, we did not want to have students from university internships distort the results: They are usually inexperienced and the code either never goes into production at all, or, if it does, the ConQAT core team will usually change it heavily.

3. Too Few Sampled Issues
   At around 100 issue per sampling group, one could argue that we did not observe a large enough sample size: It could be that we do not have enough issues to gain a representative sample of the issues. However, as a comparison of the ConQAT Random and ConQAT 100 samples shows, key metrics like the top-level category ratio and the Q-Q plots are very similar, which speaks strongly against this assumption. Furthermore, with over 300 observed issues and over 1200 categorized review

changes our study is—to the best of our knowledge—the largest manual assessment on reviews thus far.

There is a threat to the internal validity of our study that we could not fully mitigate: If communication on issues happened outside of the formal review process and tools, this probably decreases the number of review rounds needed, and could lead to some "self-motivated changes" that are in reality the suggestion of a reviewer. In ConQAT, LeVD explicitly forbids such communication, but we observed it several times during our stays at CQSE GmbH. Whenever we detected severe deviations from the prescribed processes, we rated the issue invalid, in the hope to exclude the threat. However, this process in itself could exclude certain types of issues, and therefore lead to a biased preselection of valid issues. We had no evidence to assume this in practice, though.

External Threats

External threats concern the problem of how generalizable the results of our studies are. Our case studies are exposed to three main threats:

Selection of Study Objects

By performing our case studies on two actively developed real-world OSS projects, we are confident that the results could be similar for the plethora of OSS projects which use continuous code reviews. The fact that ConQAT and GROMACS show similar results for RQs 1 to 3—despite the fact that the systems share few similarity otherwise—further supports this theory. However, every real-world system is different, and therefore we strongly assume that idiosyncrasies like the `E_META` defect categories in GROMACS, would show up for many projects in practice. This—and prior research excluding [ML09]—speaks against the idea of a "naturally given", fixed ratio of evolvability versus functional defects that code reviews find.

Subjective Defect Categorization

The categorization process for building up our database is subjective because an individual does the rating. The results are only generalizable if a high enough interrater reliability is given. We addressed this problem with two surveys which measure the amount of agreement between the study participants and our own reference estimation with the κ measure [Coh60]. Our topology is a slight adoption of [ML09], who built their topology on existing prior defect categorization that have proven to be relevant and distinguishable. [ML09] give a Cohen's κ between the two authors of the paper of 0.79, which indicates "very good agreement" between raters. Since we altered their topology, and introduced the motivation for a review change as a new concept, we have to validate that both topologies are repeatable among different raters anew.

Is the categorization done by the author of this thesis repeatable among different raters?

Survey Design To address the question whether others can replicate our estimation of the defect types and motivations, we designed two surveys:

Survey A consists of 118 questions. In each question, we ask the participant to choose the—in his opinion—best fitting defect category of a clearly marked change between two versions of source code, according to the topology from section 2.2.

Survey B consists of 17 questions. In each question, the participant shall determine the motivation for a clearly marked change in two versions of source code, according to the topology from section 2.2.

As a preparation, the study participants had access to four resources.

1. The topology overview chart from figure 2.3 (without colours).
2. The detailed description of defect categories from appendix A, with a few additional explanatory notes.
3. A description of the three motivational categories similar to section 2.2
4. A 20-minute video on how we rated defects in ConQAT with the help of our Eclipse Plugin.

We designed the time to browse through the preparation material to take one hour, and the participation in both surveys to take two hours.

We asked four computer science students in their master to complete both surveys, all of which had been studying for more than four years. Although none of the students had a scientific interest in reviews (and therefore did not know the defect topologies beforehand), three are long-term contributors to ConQAT and as such, had practical experience with reviews. Participant C had less experience with code reviews and Software Engineering in general.

We used our own estimation of the categories as the reference, and subsequently calculated Cohen's Kappa for every study participant and our reference estimation.

Survey Results Figures 5.14 and 5.15 show the results of our surveys on interrater reliability. Our results generally indicate that interrater reliability is given: The κ values on the motivation for a change are extremely high, with two raters who were in perfect agreement to our own ratings. The values on the exact defect categorizations are smaller. According to the arbitrary guidelines of [Fle81] on the interpretation of κ, they show a "fair to good agreement" at $0.45 \leq \kappa \leq 0.62$. The equally arbitrary guidelines from [VG05] would interpret the values from 0.45 to 0.60 as moderate agreement, and the values above 0.60 as substantial agreement. If we sub-summarize the same ratings as either functional or evolvability-related, the κ values for the two-top level categories again show "excellent" agreement except for participant C, shown on the right-hand side of figure 5.14 (with only two categories, the possibility for error naturally increases, therefore the greater error bars).

Therefore, we can assume the separation whether a change has functional or non-functional implications is relatively clear and largely shared among raters. This is an implication that is not evident, as we assumed it to be difficult to assess based only on the source code, whether a change had functional implications. Participant C was weakest in making the separation between a functional and an evolvability change. However, he performed well with regard to the precise defect categorization. This means that he was either

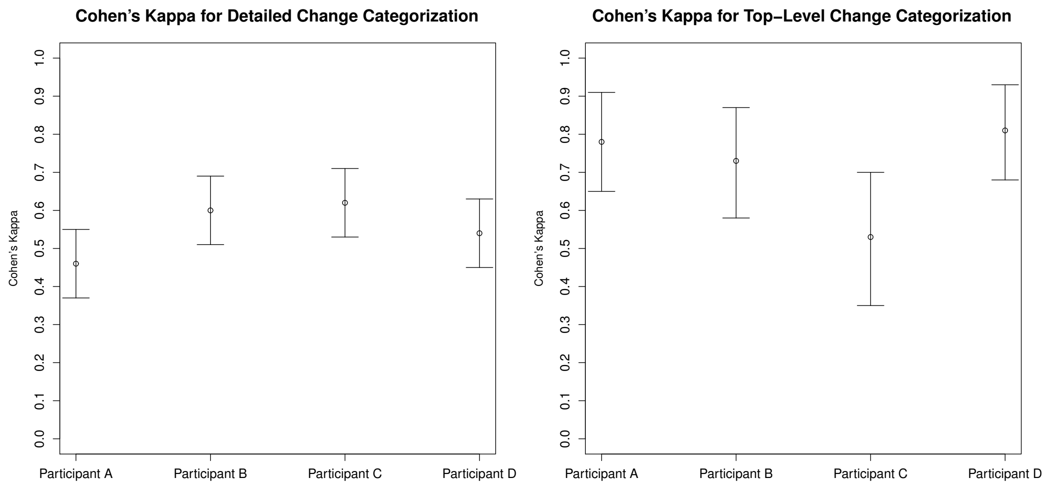


Figure 5.14: Survey A: Cohen's κ for the four Participants A–D in survey A about the defect categorization of changes. On the left hand side, we report κs for a detailed category-per-category evaluation. On the right hand side, we only distinguish between the two top-level categories evolvability and functional.

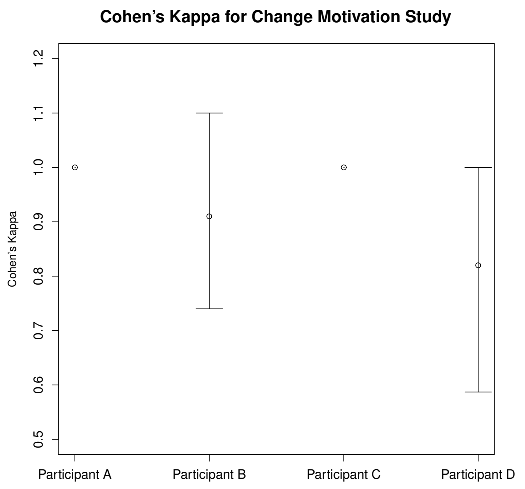


Figure 5.15: Survey B: Kohen's κ for the four Participants A–D in survey B about the motivation for changes. Participant A and C were in complete agreement with our reference categorization of the change motivations, and therefore there is no error bar.

off completely, i.e. not even hitting the right top-level group—which is not worse for the unweighted κ than rating an `E_S_O_STATEMENTISSUES` defect `E_S_O_COMPLEXCODE`—or hit the correct classification with high precision. We find a top-down approach—first establish the correct top-level group, then refine—more sensible, so we think this supports the thesis that the general concept of the proposed defect topology is well understood.

For all raters excluding participant C, the result says that while raters generally agreed on the top-groups, their agreement was not as high when it comes down to the category level. We can explain this with the multitude of similar categories which require precise reading to differentiate them. Examples of categories which are difficult to differentiate include `E_S_O_DUPLICATION` vs. `E_S_S_SEMANTICDUPLICATION`, and `E_S_O_STATEMENTISSUES` vs. `E_S_O_COMPLEXCODE`. Moreover, known as the "Kappa Paradox" [VG05], low κ values do not need to indicate poor agreement, if the categories to be rated are rare. It could be argued that this is the case for many of our categories in our survey (the average number of occurrences per category is 2.7 in survey A).

Given the relatively short preparation time, we expect that we could reach higher inter-rater reliability by allowing a longer preparation time, and providing personal trainings. The fact that participant C has only little experience with reviews, and performed worst in the rating of defect categories, supports this assumption.

The study leaves out the problem of determining the scope of a change. We cannot measure this with Cohen's Kappa, because it assumes a fixed number of values to rate. However, changes triggered by a TODO statement are trivial to spot and can make up for more than 80% of the total changes (cf. section 5.4). Therefore, even if the recognition of all other changes was off by 50%, there would still be a considerate agreement on the number of changes of more than 90%.

[EW98] investigates the repeatability of code defect classifications in even more detail. Our functional defect types are a subset of the types they suggest. Besides more advanced studies, they report κs of 0.66 to 0.82, similar to our results. This is in alignment with our results, confirming that different raters can recognize code defects reliably in a similar way.

5.6 Discussion

In this section, we interpret the collective results from RQ 1 to 3.

In RQ 1 the graphs from both ConQAT samples are similar with respect to the categories of defects eliminated during review. Therefore, the argument that the review was more shallow seems not conclusive: We would then expect to find fewer defects in categories that are difficult to fix. A better explanation could be that the relatively low number of defects stems from the fact that a well-rehearsed team of two developers (Hummel, Heinemann) did most of the work in this period.

The fact that ConQAT 100's and GROMACS's median is 0 implies that more than 50% of issues passed review directly in the first round. This could be indicative of a more relaxed review policy, or a team of authors and reviewers that is well adapted to each other's working and reviewing style. However, since the extreme outliers reported above are not due to inaccuracy in measurement, but a real-occurring situation in systems, it is generally very difficult to forecast the number of defects precisely without defining a fine-grained model that takes into account all influences to the review process. We develop such a model in chapter 6.

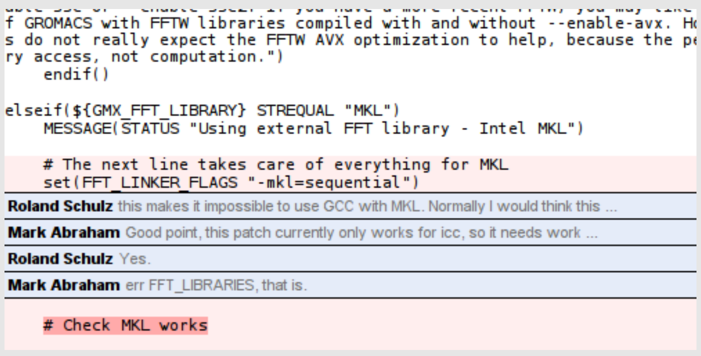
The results from RQ 2 could indicate that reviews are likely to find more evolvability than functional defects. Therefore, reviews would be especially useful if systems are long-lived and maintainability is important. [SV01] have a similar conclusion, proposing “the focus of code inspections should be expanded from just detecting defects to improving readability.” However, some researchers consider functional defects more severe in that end users of the products will likely notice them in the form of a bug, an incomplete or counter-intuitive feature. Consequently, we can still argue that reviews lead to a substantial, non-insignificant amount of functional changes in the software. Moreover, this observation was not only qualitative, but quantitatively very similar in our two systems and the two systems examined by [ML09].

Overall, we can confirm the 75:25 ratio between evolvability and functional defects reported by [ML09]. Priors works to [ML09] on different systems came to strongly diverging ratios.

Despite stark differences in the absolute number of findings, ConQAT Random and ConQAT 100 have almost identical percentage values for the three measured change motivations in RQ 3. This is an observation we could similarly establish for RQ 1 and RQ 2. This result could therefore be a further indication of the relative similarity between the two samples.

In GROMACS we have a higher percentage of disregearded and self-changed defects. The relative high number of disregarded issues could be expression of a discussion-rich review culture. Tools like Gerrit could support such a culture because they make discussing easy. In contrast, in ConQAT the discussion would have to take place in the source code, which involves starting the IDE, performing an SVN update, editing and saving the file, and then re-committing. This tedious process could hinder discussion culture on source code. On the other hand, we know from entries in the CMS and from observations at CQSE GmbH that author and reviewer discussed controversial code reviews in personal meetings.

Example 12 A discussion about a specific line of code in Gerrit.



```
f GROMACS with FFTW libraries compiled with and without --enable-avx. H
s do not really expect the FFTW AVX optimization to help, because the p
ry access, not computation.")
    endif()

elseif(${GMX_FFT_LIBRARY} STREQUAL "MKL")
    MESSAGE(STATUS "Using external FFT library - Intel MKL")

    # The next line takes care of everything for MKL
    set(FFT_LINKER_FLAGS "-mkl=sequential")
```

Roland Schulz this makes it impossible to use GCC with MKL. Normally I would think this ...

Mark Abraham Good point, this patch currently only works for icc, so it needs work ...

Roland Schulz Yes.

Mark Abraham err FFT_LIBRARIES, that is.

```
    # Check MKL works
```

To be able to compare the results from RQ 3 with the values presented in the literature, it is paramount to understand that we include a type of changes in our results that is not present in the literature—self-motivated changes. We assume that self-motivated changes were either not allowed, or not counted. Therefore, without considering self-motivated changes, ConQAT Random has 93% review-triggered and 7% discarded defects. ConQAT 100 has 92% and 8%, respectively. GROMACS has 74% and 26%. The results clearly confirm that reviewer-triggered changes lead to the majority of changes during review—as expected—, but that self-motivated changes play an important role that depends on the system.

Generally, our results to RQs 1 to 3 indicate that continuous modern code reviews focus on maintainability problems. The most frequent defects fixed are trivial naming and comment defects which typically do not require an indepth understanding of the code. Maintainability defects that require a deep understanding of the code (`E_STRUCTURE_*`) appeared in fewer numbers. It is important to acknowledge that bad variable naming and outdated comments could potentially affect code maintainability as much as bad design [SHJ13].

ConQAT reviews consistently focus on easier maintainability problems, and the distribution of defects is very similar across samples, even though recent reviews found substantially fewer defects. GROMACS on the other hand has more shallow reviews regarding the number of review findings, and concentrates its maintainability efforts even more in the easy-to-spot textual domain: Fewer substantial refactorings are suggested. However, meta evolvability data, which enables requirements and issue traceability, is a very important concern to GROMACS developers. Furthermore, GROMACS fixes a greater amount of functional defects in reviews.

6 Analysis of Influences on Reviews

In this chapter, we propose and refine a model to uncover influences on the review process. We determine which factors have the greatest impact on the outcome of a review. Concluding the chapter, we evaluate the model in practice with a case study on ConQAT.

6.1 Research Question

We repeat here the research question from the introduction.

RQ 4 *Of which kind and how strong are the influences on the number of changes or review rounds?*

6.2 Study Design

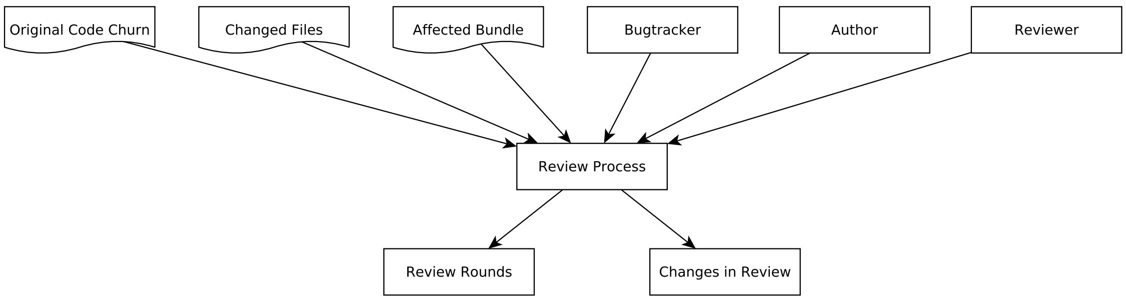


Figure 6.1: A model describing influences on the review process, and which outcome measurements they affect.

We propose figure 6.1 as a descriptive model for the influences and outcomes of the review process. To evaluate the model in practice, we transform it into a regression model. Regression models apply because they describe characteristics of a dependent variable Y (here: the review rounds, and the changes in the review) in terms of explanatory variables $X_1 ... X_n$ (here: original code churn, ...). In a more formal syntax, figure 6.1 is written as:

```
  (NumberOfTodos + NumberOfRounds) ~ CodeChurn +
NumberOfChangedFiles + Tracker + MainBundle + Author + Reviewer
```

This regression model shall be applied to each issue separately. We have to aggregate the affected variables' values on a per-issue basis. There are six explanatory variables:

- Code Churn (discrete count variable) $\in [0; \infty[$
  Code churn is a metric of how many textual changes occurred between two versions. Our assumption is that the larger the code churn in the original file, the more there is to review and therefore, the more TODOs and review rounds will follow.

- Number of Changed Files (discrete count variable) $\in [0; \infty[$
  Our assumption for including the number of changed files is that we think the more wide-spread a change is, the more concepts it touches in a system. It is difficult to master of all these concepts, and thus more TODOs would be present in issues that altered many different files.

- Tracker (categorial variable) $\in \{uncategorized, adaptive, corrective, ...\}$
  Tracker describes the type of work that is expected to occur in an issue according to [HR90]. "Corrective changes are modifications to existing functionality, while perfective changes introduces [sic!] new functionality to the system. Adaptive maintenance aims at adaptation of a system to changes in the execution environment, while preventive takes actions that will simplify or remove future, additional change requests." [RA00] Uncategorized is the default category for issues that do not fit in any of the aforementioned categories. ConQAT developers set the tracker manually in the CMS when creating an issue. We assume, for example, that corrective issues might have a lower TODO statement rate and may need fewer review rounds because they only slightly modify existing code.

- MainBundle (categorial variable) $\in \{edu.tum.cs.conqat.ada, ...\}$
  ConQAT is internally structured into more than 30 different bundles. 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 an issue.

- Author (categorial variable) $\in \{bader, beller, besenreu, deissenb, ...\}$
  The author of the original code. We could imagine certain authors being prone to receive more TODO comments than others.

- Reviewer (categorial variable) $\in \{heinemann, hummel, juergens, ...\}$
  We assume that the reviewer has one of the largest influences on a review, since the TODO comments are his work. We could imagine some reviewers to be more strict with generally more TODOs than others.

We can gather both the dependant and the explanatory variables in figure 6.1 with automatic tools. Therefore, we have to design algorithms for the automatic sampling of the metrics. Thanks to an automated collection we can include all issues in ConQAT in this case study.

6.3 Study Object

Our study object is ConQAT. For a detailed description of ConQAT, cf. chapter 4. In total, we sampled 973 issues with 2880 TODO statements.

6.4 Study Procedure

In this section we describe in detail how we carried out the study design: We introduce the algorithms used for the automatic sampling, and then continue with how applied and refined our GLM to the data.

Algorithms

Here we describe how we designed the data sampling algorithms.

Algorithm for Round Detection

The algorithm for the number of review rounds works on the commit sequence of the issue as stored in Teamscale. It first establishes the original author and the reviewer based on the first and last commit made in the issue. The implication is that the first commit is made by the author and the last commit by the reviewer (for closing the issue—i.e. making the files green). The issue is considered invalid, if the author and the reviewer are equal.

The algorithm uses a finite state machine (FSM) internally to keep track of the review rounds, as depicted in figure 6.2. For each review round there are two states: A reviewable state, in which we gather all the commits that lead to a reviewable code version, and a reviewed state, in which we gather all the commits that lead to a reviewed code version. The determination whether a commit belongs to the reviewed or reviewable commits is based on the committer. If it is the author, the commit must be in the reviewable state, if it is the reviewer, it must be a reviewed commit. If it is neither from the author, nor from the reviewer—this means a third person has commited into the issue—we leave the state machine in its current state.

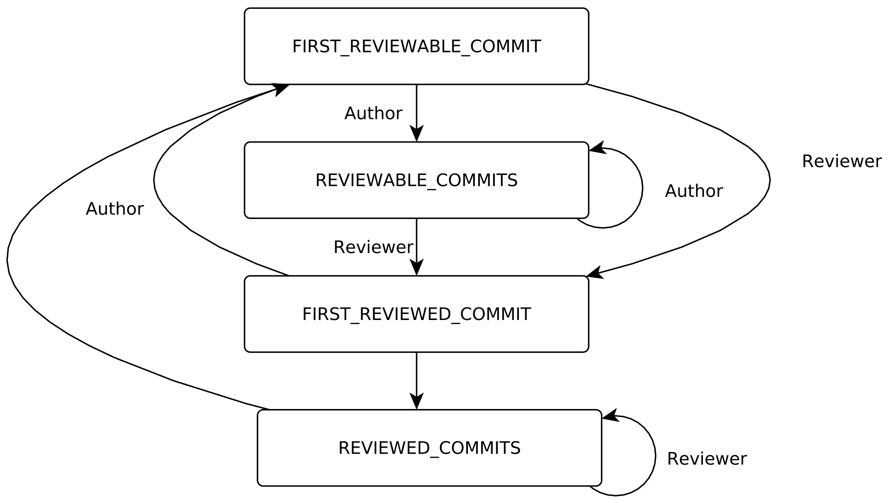


Figure 6.2: The finite state machine-based algorithm for detecting the number of review rounds.

This procedure is necessary as the review and rework process step can comprise several commits. A review round is counted as finished, when the FSM is in either of the `REVIEWED` states, and no further commits to these states, so that the next state the FSM assumes is `FIRST_REVIEWABLE_COMMIT`.

Algorithm for TODO Detection

Based on the separated review rounds, we calculate how many TODOs the reviewer added in each round. We identify the first commit and last commit of the review round—these might be the same. We then calculate a delta between the commit prior to the review round and at the end of the review round. The number of TODO statements that the delta returns is the number of TODO statements added in this review round.

Algorithm for Code Churn

For the calculation of the code churn we use ConQAT's diff algorithm, which is based on [Mye86]. We calculate the churn on the source lines of code, meaning that a code churn value of 1 represents the change, the addition, or removal of one line of source code in a file.

The code churn is calculated on the original commits: We checkout the files affected by the issue before the first commit and after the last commit in the original code version. Based on the two versions, we calculate the code churn.

Algorithm for Main Bundle

We list all touched files in the issue, and then extract from each fully-qualified file path the package. We count how many files reside in each of the extracted bundle names. The bundle with the highest number of touched files is considered the main bundle of the issue.

Algorithms for Author, Reviewer, Tracker

We extract the information for author, reviewer and tracker directly from Redmine. We then perform a sanity check whether the author and reviewer determined by the algorithm for round detection equal the information from Redmine. If they differ, we invalidate the issue.

Invalidating an issue

We designed the algorithms in such a way that they rate an issue invalid once they detect any deviations from assumed fail-safe defaults. This lead to 973 valid out of 1558 issues with contributions to the SVN.

Appliance of a Regression Model

Here we describe how we evaluated the model with the help of the statistics software R [R C13].

In order to decide which distribution for the standard count models approximates our dependant variables best, we analyse the histograms of our dependant variables. The number of review rounds is not a suitable dependant variable because it has only a very narrow range of values it can assume. Additionally, the distribution is left-skewed, since most issues have only one review round.

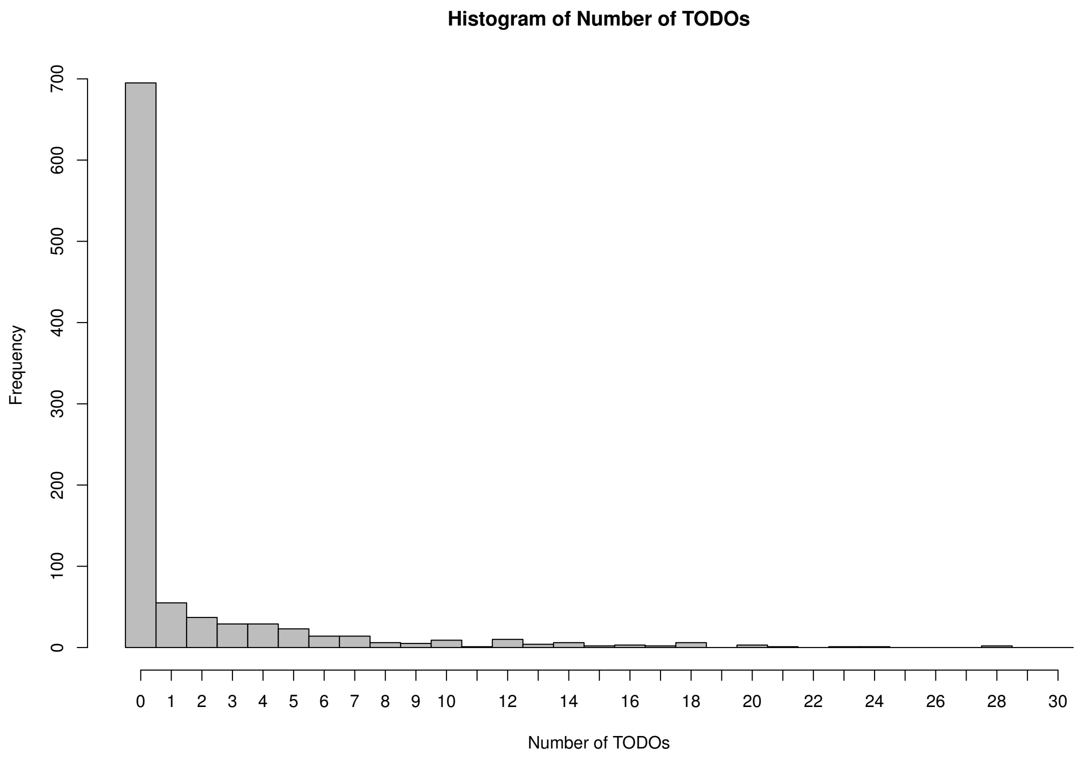


Figure 6.3: The histogram of the number of TODOs per issue.

The histogram for the distribution of the TODOs in figure 6.3 looks similar to a Poisson distribution, but is zero-inflated and skewed to the left. The minimum NumberOfTodos is 0, the maximum 164. Its median is 0, the mean 2.96. The first quantile is 0, the third 2.00.

A generalised linear model (GLM) is the preferred approach for such non-normal distributions [WH11]. We modelled the dependent variable NumberOfTodos with a negative binomial distribution. Figure 6.3 does not imply a Poisson distribution, the standard way distribution for count models. An exemplary GLM with a Poisson distribution showed strong signs of overdispersion because the varation in the NumberOfTodos histogram is greater than its mean. Consequently, this GLM yielded a very small p-value, which further indicates that a negative binomial distribution would have a better fit than a Poisson distribution [Kre99].

Effect of the Independent Variables on the Dependent Variable

In order to better understand the model, we must evaluate the relationship between each of the independent variables on the dependent variable in the model separately [Fah]. Most plots from the explanatory to the dependent variable are very diffuse. In this section, we only report on interesting relationships.

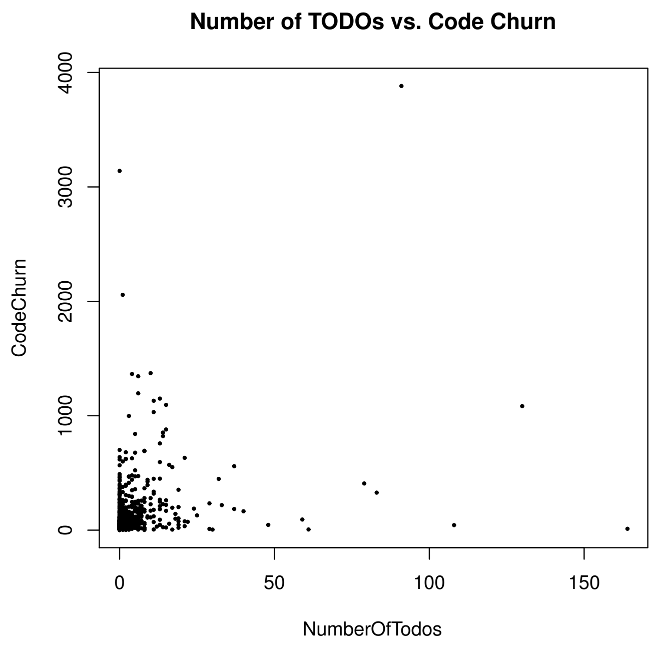


Figure 6.4: The number of TODOs vs. the code churn.

The only clear moderate correlation holds between the number of changed files and the number of TODOs: The number of changed files is linearly correlated to the number of TODOs with a significant Pearson's r of 0.65 [GN96]. However, the plot of the two variables is similar to figure 6.4, so even this strongest of all relations does not suggest an immediate linear relationship.

There are five outliers in figure 6.4, which we manually checked for validity. In issue 1251 the author removed the complete source code of JUnit from the repository and instead uploaded an archive that contained the code. This caused lots of file changes, but no TODO commments. Issue 3714 moved the LeVD rating support to the `org.conqat.engine.commons`, which caused lots of file moves, but only few TODO comments. Issue 4273 is an issue which a new student from the three weeks internships on ConQAT worked on. Issue 4387 is a large-scope bug that introduces live evaluation for the architecture editor. In the context of issue 3232, the work-intensive restructuring of ConQAT scopes was performed.

None of the outliers seems to be the result of erroneous measurement. We could debate whether to exclude issue 1251, since it is a special situation that is unlikely to re-occur. However, the measurements for issue 1251 are correct, so we decided to keep it in the dataset.

An intuitive assumption is that, as our original code churn becomes larger, we receive more TODOs. However, the relationship is surprisingly small and its plot in figure 6.4 does not suggest a linear relationship (Pearson's $r = 0.30$).

Inter-Relationships Between Independent Variables

The theory of GLMs dictates that no strong or trivial relationship in-between the independent variables should exist. In the following, we calculate Pearson's r as an estimator of the relationship between explanatory variables that are likely correlated [Fah].

The number of changed files and the code churn are related to each other. However, it is unclear how strong this relationship is from a sheer logical point of view. If the correlation is only weak, our GLM would still be working and could contain both as independent variables. Pearson's r for the number of changed files and the code churn is 0.33, which is only a mild correlation: Substantial parts of the code churn are not explainable by the number of changed files, and therefore it is valid to leave both variables in the model [Fah].

Refinement of the GLM

Our first model-fit reported a $\theta = 0.4124$, a standard error of 0.0328 and a log-likelihood of –1487.5 (2· log-likelihood –2975) [AL06]. The detailed coefficient results showed that we did not have one reviewer or author instantiation for the indicator variable reviewer and author that could report a statistically significant value. Therefore, it stands to reason whether the reviewer and author parameter as a whole have an overall significant impact on the dependant variable.

We performed a 14 degrees of freedom χ^2-test to compare the model with and without the reviewer as an explanatory variable [GN96]. The model without the reviewer had a slightly smaller θ and slightly smaller log-likelihood (cf. appendix B). At $Pr(\chi) = 0.23$ the χ^2-test implied that the reviewer variable is a statistically insignificant predictor of the number of TODOs in our model, because it is larger than our significance interval of 0.05. As a result, we refined our model to exclude the reviewer.

To test whether the author is significant in the new model, we perform a 23 degrees of freedom χ-test on a model with and without the author as an explanatory variable. At $Pr(\chi) = 0.00018$ the author does have a significant influence in the model.

A 4 degree χ-test indicates that the tracker is a statistically significant predictor of the number of TODOs ($Pr(\chi) = 3.5 \cdot \exp(-10)$).

6.5 Results

Figure 6.5 shows the refined influence model excluding the reviewer and the number of review rounds. The parameters in green boxes are statistically significant on a 95% significance interval. For the detailed results of our GLM fit, cf. appendix B. A mathematical expression for our model is given in equation 6.1, where i is the issue number [AL06].

$$\begin{aligned}\log(\text{Number of TODOs}_i) \quad = \quad & \beta_0 + \beta_1 \cdot \text{Code Churn}_i + \\ & + \beta_2 \cdot \text{Number of Changed Files}_i + \\ & + \beta_3 \cdot \text{MainBundle}_i + \beta_4 \cdot \text{Tracker}_i + \beta_5 \cdot \text{Author}_i \quad (6.1)\end{aligned}$$

One should read the parameter coefficients for count variables $\beta_1 ... \beta_5$ from equation 6.1 the following way: For example, `NumberOfChangedFiles` reports a parameter coefficient value of 0.048253. This means, that for every one unit-increase in `NumberOfChangedFiles`, the expected log count of the dependent variable `NumberOfTodos` increases by 0.048253. In other words, for every file that we touch in an issue, we expect the log of the number of TODOs to increase by 0.05. With $p < 2 \cdot \exp(-16)$, the parameter `NumberOfChangedFiles` is highly significant.

Since the log(Number of TODOs) changes, it can have greater effects than a simple linear function. This is a property of every model with an underlying logarithmic relationship.

Example 13 Given an expected number of 5 TODOsfor an issue: $\log(5) \approx 0.698$. Controlling for all other variables in the issue, we change ten additional files in the issue. $\log(\text{New number of TODOs}) \approx 1.698 + 10 \cdot 0.048253 = 2.181$. Solving to `new number of TODOs`, the number of expected TODOs rises from 5 to 8.9.

Example 14 Imagine an issue 1 that implements a new feature (tracker perfective), and which has an expected 20 chanes. Imagine a corrective issue 2 identical to 1 wrt. all other variables: $\log(\text{Changes}_2) \approx \log(20) - 1.365 = 1.63 \Rightarrow \text{Changes}_2 \approx 5.1$. We expect issue 2 to have 15 changes in review less than issue 1 (75% reduction), and the only reasons for this is a change in the tracker.

One has to interpret categorial variables like `Tracker` slightly differently: Since it is the default value for this category, the value `uncategorized` in the tracker is missing. All other values like `corrective` are relative to the value of `uncategorized`. Thus, if one sets the tracker to `corrective` for an issue, we expect 0.65 fewer log TODOs than for `uncategorized` issues. The other categorial variables follow this interpretation scheme.

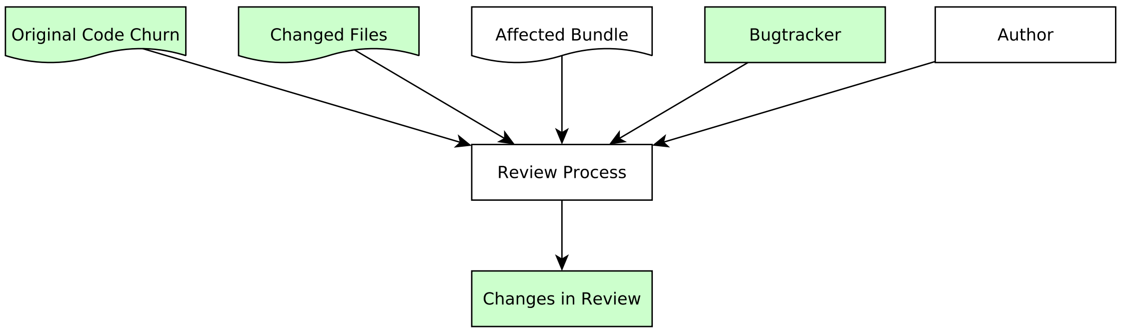


Figure 6.5: The refined model of influences on the review process.

We used the valid issues from both ConQAT samples to generate one large, coherent database which we could use as reference to compare the automatic review findings against. Figures 6.6 and 6.7 show the manually assessed values versus the automatically determined values for the number of rounds and the number of changes per issue. A triangle symbolizes the manual measurement, and a dot the automatic data. We observe in both graphs that the automatic measure is almost always either spot-on for the majority of measurements, or a small under-estimation of the manual assessment. Therefore, the number of TODOs seems a good estimator of the number of actual changes in an issue, as shown in figure 6.7. This is not surprising as we observed in RQ 2 that 12% of changes in ConQAT are self-motivated, and 7% are discarded, so they roughly cancel each other out.

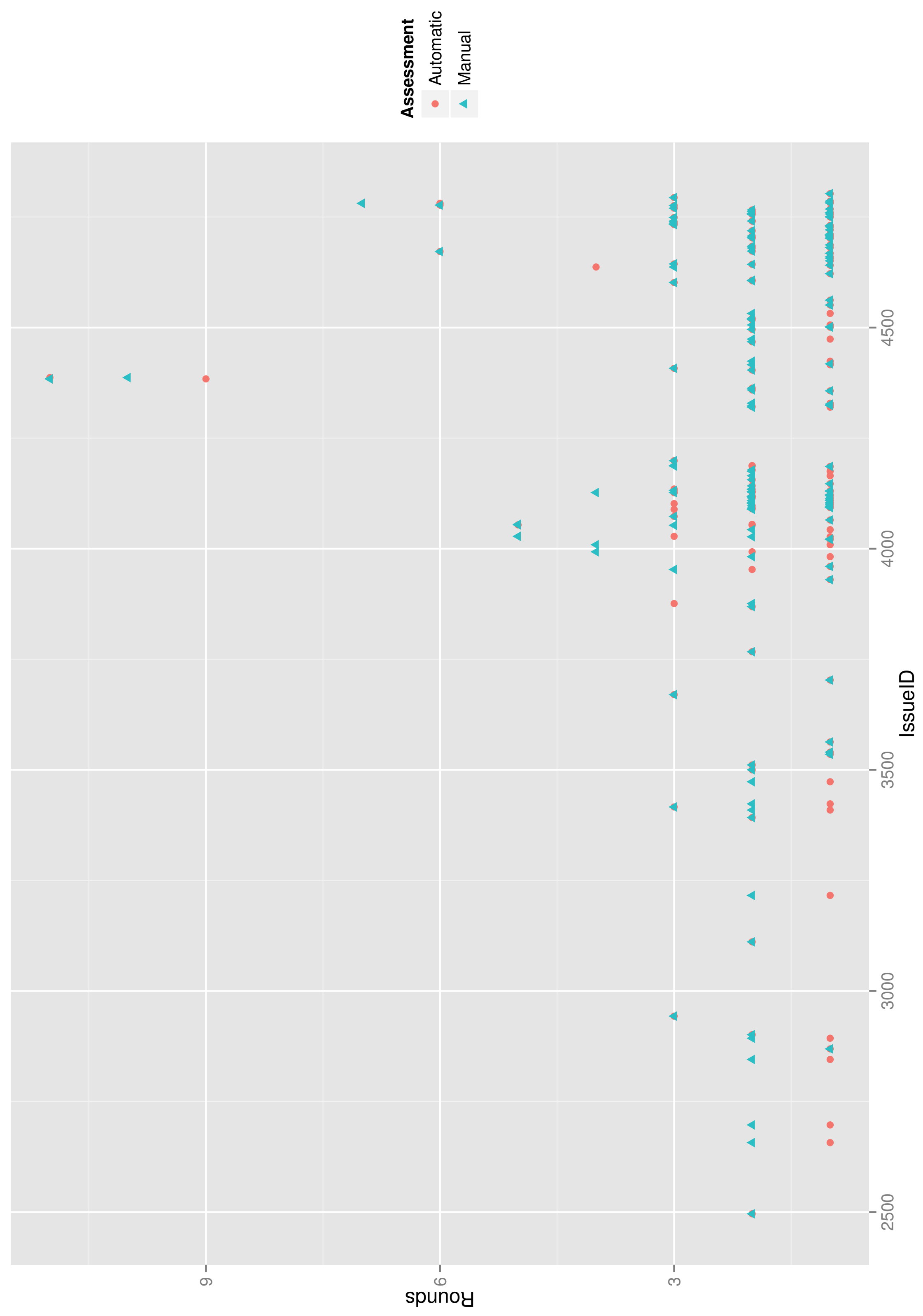


Figure 6.6: The number of manually assessed rounds from the combined ConQAT samples plotted against the number of automatically determined rounds per issue.

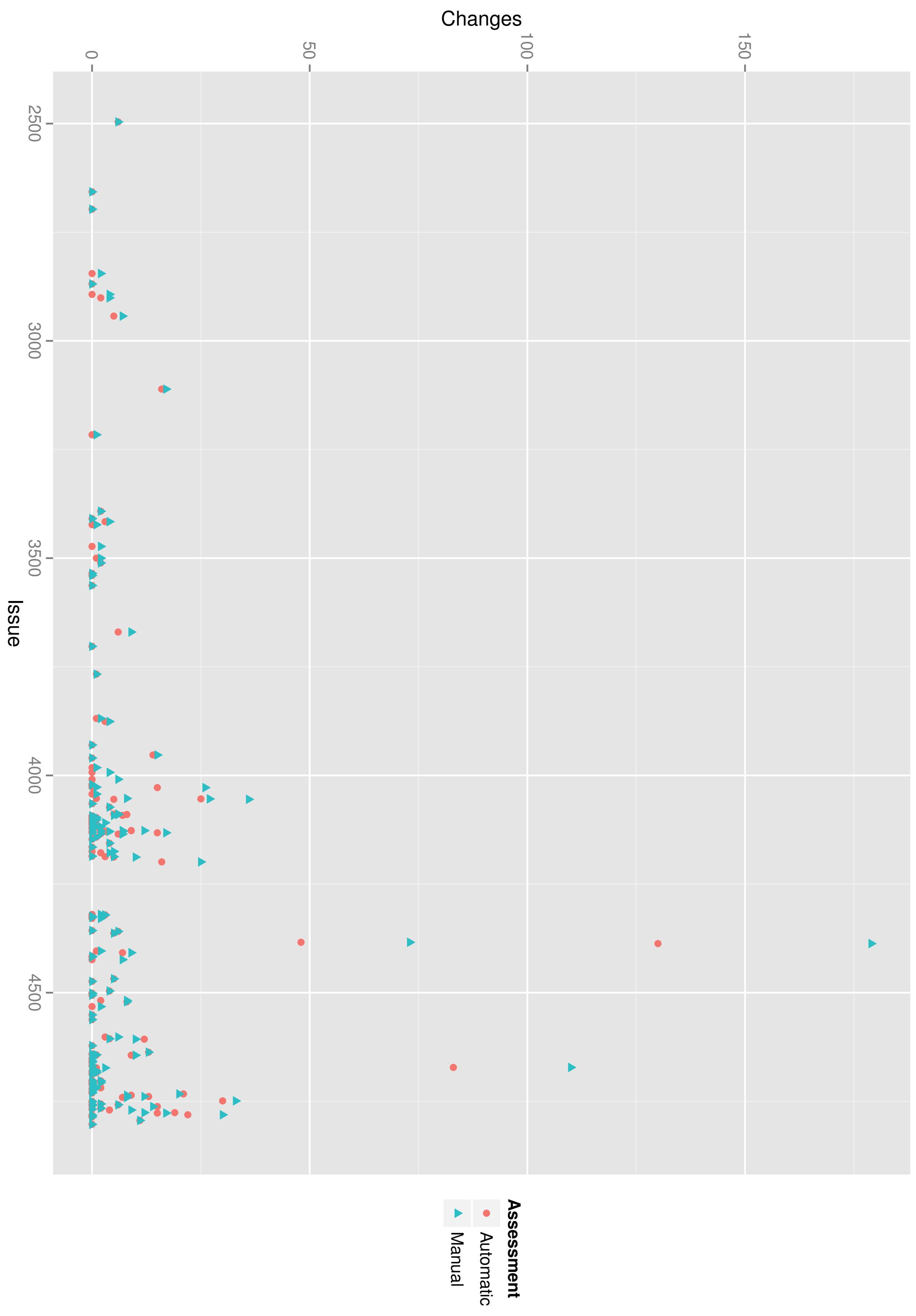


Figure 6.7: The number of manually assessed changes from the combined ConQAT samples plotted against the number of automatically determined TODOs per issue.

The results report a $\theta = 0.400$, a standard error of 0.0317 and a log-likelihood of –1496.5 (2· log-likelihood –2993). We test for goodness-of-fit of the model to the data with a χ^2 test on the residual deviance and degrees of freedom [GN96]. The residual deviance of 763.27 on 867 degrees of freedom is highly insignificant $1 - pchisq(763.27, 867) = 0.995$, which means that the negative binomial distribution fits the data well.

However, many of the parameters in the result show a high variability and therefore do not lie within standard confidence intervals. Reasons for this are mainly the left-skewed TODO histogram (cf. figure 6.3), and the fact that there is no strong relationship between any of the explanatory and the dependent variable (cf. section 6.4). Therefore, results from the GLM should be considered with care, even though the model has a high goodness-of-fit.

Both our non-categorial count variables are significant: The number of changed files has a coefficient of 0.048, and the code churn of the original commit a coefficient of 0.0025.

6.6 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.

Even though we designed our algorithms to assume safe defaults and skip an issue when they detect problems, there is the risk of a systematic failure for the review round and number of TODO detection algorithms. We used the valid issues from both ConQAT samples ConQAT Random and ConQAT 100 from chapter 5 to generate one large, coherent database which we could use as reference to compare the automatic review findings against. As shown in figures 6.6 and 6.7, our automatic approximations are quite accurate.

Issue	Review Rounds		Changes	
	ConQAT Random	ConQAT 100	ConQAT Random	ConQAT 100
4118	2	2	2	1
4127	4	3	12	7
4129	2	2	7	4
4703	2	2	2	2
4741	2	3	8	8

Table 6.1: A comparison of key metrics of the issues that we sampled independently for both ConQAT Random and ConQAT 100.

Table 6.1 depicts five issues which we sampled independently in both ConQAT 100 and ConQAT Random. When the two samples did not agree, we assumed the lowest reported number. While most issues are similar, issues 4127 and 4129 have quite differing change counts, although the same person sampled the issues. We discuss this problem of observer reliability and how we mitigated it in section 5.5.

Some instantiation of variables have too few values. Therefore, we could exclude these from the model. However, we rely on the fact that they will not be significant and leave them in the model. Consequently, we can only interpret significant data.

Given the distribution of TODOs in figure 6.3, one could argue that we should use a two-step hurdle model, or a zero-inflated model. Both models do not apply in our context: Hurdle models describe the dependent variable as an outcome of a two phase processes, which is not the case for the number of TODOs statements: Reviewers do not flip a coin to decide whether they will write any number of review comments, or none [Gre94]. Zero-inflated models attempt to account for excess zeros. They assume there are two kinds of zeros, correctly-measured zeros and false excess zeros, called structural zeros [BZ05]. However, our zeros are correctly measured and part of the data. Therefore, a zero-inflated model approach is not applicable for our data.

Another crucial threat is that we have not included all influencing dependent variables in our modelling. This is almost certainly true, since we only included technical, measurable aspects of reviews. Difficult to measure benefits like knowledge transfer, enhanced team spirit and communication go beyond the scope of this thesis. However, we established that our model fits the data well, so that—at least for the variables we have modelled—we find no obligations against our refined model.

External Threats

External threats are about the generalizability of our results.

While we assume ConQAT is prototypical of many current OSS projects that employ continuous reviews, the analysis of only one system does not allow us to draw conclusions about the review process in general. To mitigate this thread, we would need a larger case study on more projects.

Our model figure 6.5 contains variables that could be measured on most projects, since it is general information that almost any software project equipped with a VCS could provide: The author, the reviewer, the code churn, the number of touched files, and the number of TODOs would be similarly extract on any system. In contrast, some systems might not have a system architecture from which we could infer the MainBundle of the issue. Additionally, not all systems assign a tracker to a bug. From our knowledge of OSS systems, we believe that these systems are the majority. Nevertheless, a model without the MainBundle or Tracker could still make sense. Other projects might even provide additional information that could go into the model, e.g. whether a developer was a core developer or a newcomer.

6.7 Discussion

In this section, we discuss and interpret the results from our regression model.

A surprising finding is that the reviewer did not have a significant influence in our model. All other variables have a significant impact. We expected that certain reviewers tend to place more review comments than others, but this was apparently not the case in ConQAT. However, one cannot draw the conclusion that the reviewer does not have a substantial influence on the review result. This is obviously so, since the reviewer is the one to place the review comments. We can only say that, in our influence model on ConQAT, the reviewer had no significant influence on the number of changes. If we considered the types of the changes, we would likely have gotten a strong influence of the reviewer: In ConQAT, almost all `F_ALGORITHMPERFORMANCE` defects came from one reviewer.

The results confirm many of our initial assumptions about code reviews: As we change more files, we expect more changes in the review. This is similarly true, although the relationship is not as strong, for the code churn of the original commit with a coefficient of 0.0025: The more lines of code we originally change, the more changes we are likely to perform during review. Both coefficients lie in a significance interval of 0.001, which makes these statements very reliable.

Another initial guess was that issues in the tracker corrective will cause fewer defects, while issues in the tracker perfective will cause more defects. We can confirm the statement on a 0.05 significance interval in our model. If the tracker corrective is used, the expected log count of changes decreases by 0.65 compared to uncategorized issues. Implementing a perfective issue will increase the expected log count by 0.70. Therefore, the difference between corrective and perfective issues is a significant parameter difference of 1.365. Example 14 demonstrates what an effect the tracker has on an otherwise unchanged issue. Comparing it with other significant values, e.g. the code churn or the number of changed files, we can conclude that the tracker has a strong influence on the number of expected changes for average-sized issues.

Only few of the categorial variable instances in appendix B lie in standard significance intervals. As we established, this is not due to a lack of model fit (cf. section 6.5), but because of the high variances in the data set and the zero-inflated histogram of the dependent variables (cf. sections 6.4 and 6.6). For example, none of the author instantiations had a significant value. At significance values of 1, their coefficients are highly insignificant. This is likely the result of a great range of expected values for the number of changes per author: All authors had many issues with zero changes, but some with substantially more. This makes the author a bad regression parameter for the actual number of changes. However, it is interesting to see that all authors with a long history of developing ConQAT—Beller, Deißenböck, Feilkas, Göde, Heinemann, Hummel, Jürgens, Kanis, Kinnen, Pfaller, Poehlmann, Streitel—had similar coefficients around 33: Beller reported the highest coefficient at 35.5, while Kinnen had the smallest coefficient at 31.9. Given the lack of confidence on these parameters, we can only interpret the values as a trend that could need further studies. The basic message is that the main developers seem to perform roughly the same number of changes on average. We can only explain the lower coefficients for authors like Besenreuther and Hodaie—who were students from university internships—by the fact that for these authors, issues with a very low number of TODOs were valid, while they either did not have issues with a lot of TODOs, or those were invalidated. Since a biased selection of issues for authors with very few defects will always be a threat, we recommend to only interpret values for significant variables. For these, we are safe to assume that we sampled enough observations.

Similarly, for the MainBundle variable, we observed only two significant values. The coefficients for most bundles are within $[-1;1]$, but there are some outliers like `org.conqat.engine.server`, all focusing around –37. For these bundles, we had few observations, so that reviews with zero TODOs over-weighed. For example, `org.conqat.engine.server` has only 1 issue, `org.conqat.engine.bugzilla` 2 issues assigned to it, whereas `org.conqat.engine.commons` has 80 issues. This could be either because the bundles existed only for a short time, or because we did not sample enough observations for these bundles.

7 Conclusion

In this chapter we describe contributions and conclusions from our thesis.

We have refined and proposed a framework for the quantification of code review changes that encompasses a defect definition, a defect topology and three research questions. It bases on the novel assumption that every change in the review process can be modelled as a defect, and that the motivation of a change is an orthogonal classification. The research questions are empirically answered in case studies on ConQAT and GROMACS, which comprise over 1300 categorized defects.

Our case studies show that defect distribution is similar across systems, even though the number of findings per issue differs greatly. Documentary defects, especially changes in comments and identifier names, make up most of the defects in the systems. The more difficult to find structural evolvability defects form a minority in both systems. This confirms findings from other contemporary research on light-weight reviews.

We detected a ratio of evolvability to functional changes of $\approx$ 75:25 for both ConQAT and GROMACS. While this confirms recent research, we do not have enough evidence to assume the 75:25 ratio a "universal constant".

We examined the motivation for changes in reviews and found that the majority of changes is triggered by a review comment (80% for ConQAT, 60% for GROMACS). Self-motivated changes account for a smaller, but relevant part of all changes at 10% for ConQAT and 20% for GROMACS. This indicates that studies which did not address these changes could be biased if self-motivated changes are allowed in the review process. The majority of review suggestions is realised, and only a minority is discarded. Therefore, reviews are useful in practice to and lead to changes in the system.

These findings suggest that reviews are a sensible measure to ensure the maintainability of long-lived software systems. However, reviews might be of less value if maintainability is not a concern.

As a second case study, we created and refined a model for the outcomes and influences of the review process on ConQAT. The study is based on a database with 973 automatically sampled issues. We showed that the reviewer had no significant impact on the number of expected changes. Other intuitive assumptions about the review process turned out to be true: Bug-fixing issues produce significantly less changes than issues which create new functionality. The more code churn or the higher the number of touched files in an issue, the more changes do we observe on average. Our result indicates that the MainBundle has an influence on the number of changes. However, since only two of the values are significant, we cannot draw conclusions from it. The data regarding the effect of the issue's author was too variable, but hints at the fact that all core developers in ConQAT have a roughly similar probability for receiving the same number of changes.

8 Future Work

In this chapter we outline interesting future research work beyond the scope of this thesis.

8.1 Automated Reviews

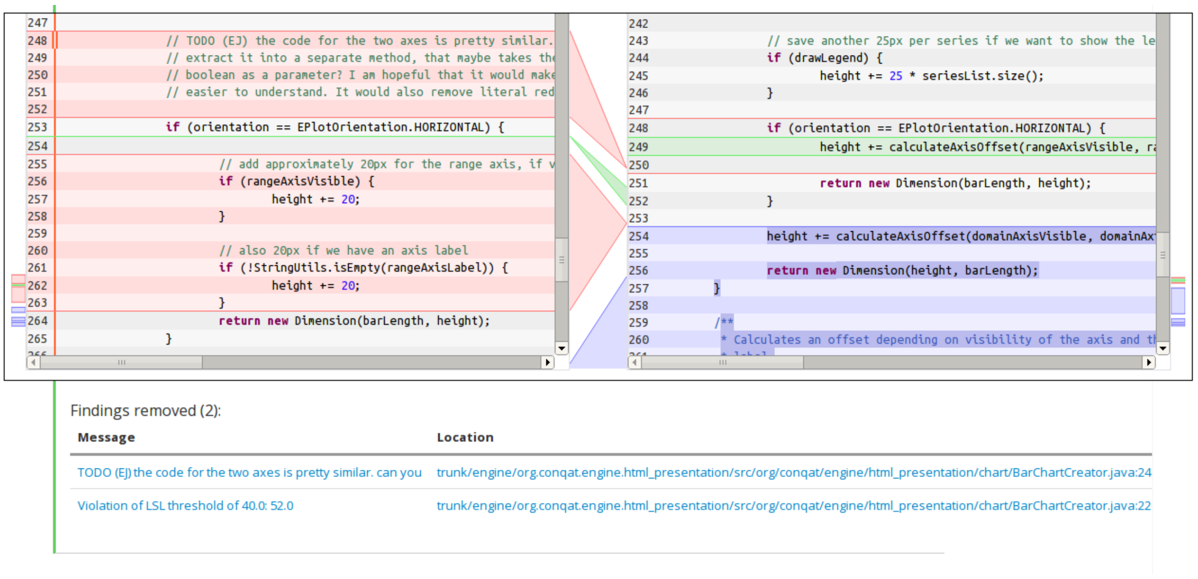
In our case study on ConQAT, we largely ignored findings in the "Visual Representation" category, as all ConQAT developers use the Eclipse IDE. Eclipse can be configured to use its integrated code formatter automatically upon saving a document. Therefore, the automatic code formatter can handle most of the visual defects—bracket usage, indentation, long line and space usage—without manual interaction.

Similarly, could automatic defect findings tools like FindBugs [Fin], PMD [PMD], FxCop [FxC] and StyleCop [Sty] find some of the reviewer's suggestions, that are less trivial than visual defects? Could this make some of the review efforts redundant?

Typical examples for review findings created by FindBugs are a method that is too long (cf. figure 8.1), or a missing null check. If the reviewer does not have to look for certain trivial kinds of defects he can concentrate on the more substantial functional and evolvability defects. The use of automated defect finding tools could result in a reduction of costs for reviews and ensure a more consistent detection for defects like null pointer exceptions. Moreover, reviews would be more consistent, as some defects would be found independent of who reviewed the code.

An open research question for a future case study would be to analyse which kinds of defects can be found by state-of-the-art defect finding tools. To address to which degree they overlap with defects found in the review, the tools could be employed on reviewed code. Judging from the percentage of overlap between automated and human findings, we could draw conclusions whether these tools can replace reviews to some extent, or should rather be used in parallel.

As an example from Eclipse's code formatter shows, these tools would currently not be able to completely replace the human reviewer: The formatter cannot perform grouping of related program code lines into one "paragraph". This would require a logical recognition of associated program lines beyond the capabilities of today's code formatter. Many blank lines—especially within functions—have to be removed manually by the developers. We would expect to find similar defects categories that automated tools can't check. Experimentation with such tools during our case studies indicated that findings from FindBugs often went hand-in-hand with reviewer findings, although the reviewer suggestions were reason-based and the automated findings only based on thresholds like maximal method length or class length. A further interesting research question could therefore be if such tools could at least effectively support the reviewer.



```
            // TODO (EJ) the code for the two axes is pretty similar.
            // extract it into a separate method, that maybe takes the
            // boolean as a parameter? I am hopeful that it would make
            // easier to understand. It would also remove literal red

            if (orientation == EPlotOrientation.HORIZONTAL) {

                    // add approximately 20px for the range axis, if v
                    if (rangeAxisVisible) {
                            height += 20;
                    }

                    // also 20px if we have an axis label
                    if (!StringUtils.isEmpty(rangeAxisLabel)) {
                            height += 20;
                    }
                    return new Dimension(barLength, height);
            }
```

```
            // save another 25px per series if we want to show the le
            if (drawLegend) {
                    height += 25 * seriesList.size();
            }

            if (orientation == EPlotOrientation.HORIZONTAL) {
                    height += calculateAxisOffset(rangeAxisVisible, ra

                    return new Dimension(barLength, height);
            }

            height += calculateAxisOffset(domainAxisVisible, domainAx

            return new Dimension(height, barLength);
    }

    /**
     * Calculates an offset depending on visibility of the axis and th
```

Findings removed (2):

Message	Location
TODO (EJ) the code for the two axes is pretty similar. can you	trunk/engine/org.conqat.engine.html_presentation/src/org/conqat/engine/html_presentation/chart/BarChartCreator.java:24
Violation of LSL threshold of 40.0: 52.0	trunk/engine/org.conqat.engine.html_presentation/src/org/conqat/engine/html_presentation/chart/BarChartCreator.java:22

Figure 8.1: The automated code finding "Violation of LSL Threshold" is in alignment with the reviewer's manual finding to extract a method. The review comment states that the method contains duplicated code and that extracting a method would increase readability. In contrast, the automated warning finds a function that is too long. The solution to both findings is to extract a method.

8.2 Comparison of File-Based vs. Change-Based Reviews

A prejudice against changed-based reviews could be that the big picture is lost over time. Once defects are present, they may go unnoticed as long as this particular part of the code is not changed. In contrast, LeVD is a particular review process in which not only the changeset—like in Gerrit—is analysed, but the reviewer is encouraged to assess all touched files thoroughly. A possible research question for such a work could be: *What are the benefits and costs of a file-based review strategy over a change-based review?*

8.3 Further Case Studies

Since we could only examine two OSS systems in this study, we need further empirical evidence to confirm RQs 1–3 on a broader basis. An interesting target would be to see if other systems also center around the 75:25 percentage of evolvability to functional defects, and why. Particularly for a comparison of the benefits of file versus change based reviews, we would need an even larger database, as there is a plethora of other uncontrolled confining variables.

We examined influences on the review process in Chapter 6 only on ConQAT. Since we already have manual data on GROMACS from RQ 1–3, an idea would be to perform a similar influence analysis. This would require us to adopt the automated assessment algorithms from ConQAT to GROMACS, addressing questions such as how to deal with multiple reviewers. Could models on other systems confirm that the reviewers did not have an influence on the number of TODOs?

Since the model from RQ 4 did not have significant values for many of its variables, it would be interesting to see if other models than generalised linear models could return smaller confidence intervals. While we do not expect fundamental changes because of the small correlation of the independent variables with the dependent variable, a mixed model approach could refine some of the parameter coefficients. This would further increase the validity and generalizability of our results.

A Review Defect Classification

Code Review Defects

Authors: Mika V. Mäntylä and Casper Lassenius
Original version: 4 Sep, 2007
Made available online: 24 April, 2013

This document contains further details of the code review defects presented in [1]. Description of each defect is given as well as notes describing the most typical cases.

[1] Mäntylä, M. V. and Lassenius, C. "What Types of Defects Are Really Discovered in Code Reviews?" IEEE Transactions on Software Engineering, vol. 35, no 3, May/June 2009, pp. 430-448, Available online:
http://dx.doi.org/10.1109/TSE.2008.71
http://lib.tkk.fi/Diss/2009/isbn9789512298570/article5.pdf

1. Documentation - Textual Defects

Defect	Naming
Description	Problems relating to software element (=methods, classes, variables, etc) names
Notes Industrial	Most of the Naming defects were due either to not conforming to the company's naming policy or having uninformative names. In some cases, naming defects led to longer discussions on how to name certain elements as the company did not have policy for naming every code element. Some naming problems were inherited from legacy code that did not conform to the current naming policy. Discussions with the industrial reviewers made it clear that the company strongly believed self-descriptive naming over code commenting explaing the difference in Naming issue shares.
Notes Student	Most of the Naming defects found by students pointed out uninformative names. Often the names were too short, or too general, e.g. "arg1" and "arg2". In a few cases, the names were incorrect when compared with the behavior. Occasionally, the naming did not follow Java coding standard by SUN.
Defect	Comments
Description	Problems in code comments
Notes Industrial	Comments were needed, e.g., to explain complicated programming logic. Some comments were incorrect, which was often the result of old legacy comments or copy-paste-programming. Some comments lacked information required by the company's commenting policy.
Notes Student	Many Comments defects were related to the omission of JavaDoc elements, e.g., missing method descriptions. Some comments were requested to further explain the rules implemented. Finally, some defects pointed out incorrect comments or just simple spelling mistakes.
Defect	Debug Info
Description	Problems with debugging messages. Debug Info is placed in this sub-group because it improves programs static and runtime documentation.
Notes	All the Debug Info defects requested more information on the error condition
Defect	Other Textual Defects
Description	Other Textual Defects that could not be placed to other defect classes.
Notes	Should comments or method prototypes be presented in the header files or in the source files, general comments of language usage affecting all comments and names.

2. Documentation - Supported By Language Defects

Defect	Element Type

Description	The software element is with wrong type (only cases not causing runtime failure)
Notes Industrial	All Element Type defects were in Return value types, e.g., changing the return value from “int” to “boolean”, or returning a value instead of a “void”. Interestingly, the different teams of the company had somewhat different policies regarding return values. In one team, “void” was allowed as a return value type, while in many others it was not permitted.
Notes Student	In the student reviews, the Element Type defects were more diverse. In addition to defects related to return value types, there were excessively general description of exceptions (e.g. throws “Exception” versus throws “IOException”), missing an implemented interface in the class description, having the wrong type of variables, claiming that a method throws an exception when it does not, and unnecessarily extending the “java.lang.Object” class as it is extended by default.
Defect	Immutable
Description	Not declaring variable to be immutable when it should have been or declaring it immutable when it should have not been
Notes	This was present in both reviews as both languages Java and C support such mechanism.
Defect	Visibility
Description	Software element (e.g. method, variable, class) has too much or too restricted visibility
Notes	In the industrial reviews, all the Visibility defects addressed method visibility, and in the students reviews there were cases of both the methods and variables visibility
Defect	Void Parameter
Description	Using empty brackets instead of keyword “void” as parameter
Notes Industrial	This was only present in the industrial reviews as Java does not have such option.
Defect	Element Reference
Description	Referring to software element with incomplete name
Notes Student	This was only present in student reviews and all defects found indicated that keyword "this" should have been used when referring to instance variables or methods

3. Visual Representation

Defect	Bracket Usage
Description	Incorrect usage of brackets or braces
Both Reviews	In both reviews these issues mostly referred to cases in which the developer had omitted brackets when having only a single statement after a conditional branch. Occasionally, brackets were requested to clarify the precedence operations in complex comparisons or mathematical statements.
Defect	Indentation
Description	Incorrect indentation
Notes	Often the developers thought the indentation might be correct in the code, and it was wrongly indented only in the printing. However, since they from the point of view of the review sessions were defects, we cannot remove them from our analysis simply because the developer believes this is not a problem in the code.
Defect	The Blank Line Usage
Description	Blank Line Usage is incorrect
Notes	This occurred mostly due to having an excess of blank lines or too few blank lines. However, it also included cases in which lines were split at incorrect positions.
Defect	Long Line
Description	Long Line means defects where a long code statement was contained in an excessively long single line, often more than 80 characters.
Defect	Space Usage
Description	Space Usage defects referred to the usage of the blank space character in the code.
Defect	Grouping
Description	Grouping refers to the grouping of code elements, e.g., in header files, one should group functions so that closely related functions are close to each other in the file, introducing class variables at the

3

	beginning of the class.

4. Structure - Organization Defects

Defect	Move Functionality
Description	Move Functionality, refers to the need to move functions, part of functions, or other functional elements to a different class, file, or module.
Defect	Long Sub-routine
Description	Function, procedure or method is of excessive length and functionality.
Notes	Some of these recommendations suggested particular elements that should be extracted to a new method while others merely pointed out that method is too long and difficult to understand
Defect	Dead Code
Description	Code that is not executed or used in the software.
Notes Industrial	Industrial Dead Code defects were evenly distributed and they consisted of unnecessary variables, unnecessary headers, uncalled functions, and branches of code that are never executed.
Notes Student	The most prominent Dead Code defect in the student code reviews was an unnecessary header with 6 mentions. Other Dead Code defects in the student reviews were an unnecessary type cast, an unnecessary variable, an unnecessary method, and an unnecessary return statement.
Defect	Duplication
Description	Code that is duplicated
Notes Industrial	The Duplication issues included functions that were partially duplicated, completely duplicated functions, duplicated comparison operations, and duplicated variables.
Notes Student	They types were similar to the issues of the industrial review. In the student code review, many of the duplicated code segments were quite small three to five lines of code, but there were duplicates up to 20 lines of code.
Defect	Complex Code
Description	a piece of code that is difficult to comprehend
Both reviews	Some Complex Code defects were quite general, e.g. a method is messy and needs to be rewritten, pointer usage is wild, or the implementation is unnecessarily complex. Other defects were more specific, e.g., an if-statement has too many comparisons or unnecessary use of negative comparison when using a positive result would be more readily understood.
Defect	Statement Issues
Description	These require splitting, combining or otherwise reorganizing a statement inside a function.
Notes Industrial	identified once in industrial reviews, when a reviewer suggested combining a declaration and an initialization of a variable to one code line
Notes Student	Six Statement issues were recognized in the student code reviews. Three of them suggested splitting a long boolean return statement to several if-else statements. Three Statements issues suggested combining statement, two of them suggested combining variable introduction with initialization and one them suggested combining statements rather than using temporary values to store the results.
Defect	Consistency
Description	Means the need to keep code consistent in a sense that similar code elements operate in a similar fashion and are more or less symmetrical. For example, similar tasks in similar classes should have similar implementations
Notes Industrial	Examples the code should not alter between using pre-defined values and numeric values, functions performing similar task in similar classes should have similar naming and naming and implementation, and comparison statements should be consistent with each other or even extracted to their own functions to remove repetition.
Notes Student	In student reviews only one Consistency issues was found. This issue pointed out inconsistent variable initialization, as some variables were initialized at the variable declaration, others at the method's constructor and some of the variables where not initialized at all.
Defect	Other
Description	Other Organization defects

Notes Industrial	In the industrial reviews, these defects included splitting up a large file containing several classes and 2000 lines of code into several files; defects were a logical piece of code was unnecessarily split into several places; the need to remove wrong couplings between software elements; the need to split up functionality into several implementations; and reducing the number of different error handling mechanisms.
Notes Student	In the student reviews, they included having several return statements in a method; using loop variables outside of the loop structure; in-lining a method as part of an other method; starting indexing from zero instead of one; using negative return values instead of positive; having too many temporary variables; having variables in class scope instead of method scope; having a method with too many parameters, and commented code that should be deleted.

5. Structure - Solution Approach Defects

Defect	Semantic Duplication
Description	Means syntactically different code blocks with equal intent, e.g., different sorting algorithms such as quicksort and heapsort have equal intent but they are not identical at the code level.
Notes Industrial	In the industrial case, we had no defects where the code under review had Semantic Duplication with itself. Instead, all the Semantic Duplication defects were identified based on the reviewers' knowledge of existing functionality located elsewhere in the software system. E.g. functionality is already implemented in some other place and there is no need to duplicate this functionality in the code under review.
Notes Student	All Semantic Duplication defects in the student reviews suggested replacing code with pre-built functionality included in the Java class library.
Defect	Semantic Dead Code
Description	Code fragments that are executed, but they do not serve any meaningful purpose and/or have no effect on the result
Notes Industrial	In the industrial reviews, two code review sessions identified nearly half of the Dead Code and Semantic Dead Code defects. This was caused by code templates that were re-used and modified.
Notes Student	All the Semantic Dead Code defects of the student reviews were unnecessary checks performed in conditional statements while in the industrial review the reasons were more varying
Defect	Change Function
Description	Need to change a certain function call to another when the program used old or deprecated functions
Notes Industrial	Often the reason for changing a function was that a better alternative was available, for example, the company had provided wrapper functions on top of many basic C-functions that added extra features or made using the functions easier.
Defect	Use Standard Method
Description	Use Standard Method contains defects where a standardized way of working should be used.
Notes	In the industry reviews these often meant using predefined constants rather than magic numbers. In the student the use exceptions for error messaging instead of return values was mentioned often. Other issue in this category were use of enumeration instead of integers; and use accessor-methods to access a class.
Defect	New Functionality
Description	Need of new functionality to ensure evolvability
Notes	In the industrial reviews, we saw defects where the creation of new functionality or classes was required to make the software more evolvable. We categorized these under the heading of New Functionality. No such defects were identified in the student reviews.
Defect	Other
Description	Other contains a wide range of defects that truly represent the Alternative Approach in its most fruitful form
Notes Industrial	In the industrial reviews, this type contained implementation changes such as using arrays instead of other more complex memory management structures, changing the code to enable an easier removal of several data items from the database, using dedicated arrays for each data element instead of a

	shared array, and using a simpler and more efficient way of keeping records in a database.
Notes Student	In the student reviews Other defects included suggesting a simpler way of performing computing and comparison operations, using Java's Generics data structures, and caching numeric values rather than recomputing them.
Defect	Minor
Description	Minor gathers implementation changes, but the defects are easier to fix and seemed less important than those categorized under Other
Notes Industrial	These defects were only identified in the industrial review. Examples of such defects are, having a default branch in a switch block, changing an if-else block to a switch-block, changing comparison element from a class name to a class id.

6. Resource Defects

Defect	Variable Initialization
Description	Means defects where variables are left uninitialized prior to use. Uninitialized variables may contain any value and using such variable for comparison or calculation produces arbitrary results.
Notes	Only present in industrial reviews because students used Java language that forces primitive variable initialization
Defect	Memory Management
Description	Means a defect where a mistake is made in handling the system memory.
Notes	In industrial reviews these include allocating too little memory, not freeing memory after its use, and freeing the same memory more than once. In the Java language used by the students, memory management is automatic, thus, the students identified no memory allocation defects.
Defect	Data & Resource Manipulation
Description	Defects related to manipulating or releasing data or other resources,
Notes	For example not releasing database cursor, and mistakes in data modification. There was only single Data & Resource manipulation defect in the student code reviews, and it is likely due to the application, which did not require many data structures.

7. Check Defects

Defect	Check Function
Description	Means that when a function is called there is also a need to check that the value returned is valid and that no error occurred
Notes	In industrial reviews 4 instances were needed to check the result of memory allocation operation
Defect	Check Variable
Description	Means that there is a need to check variable
Notes	Often such variables were function parameters, or loop control variables. In industrial reviews pointer checking was found 6 instances
Defect	Check User Input
Description	Check User Input means the need to validate user input
Notes	No additional description

6

8. Interface Defects

Defect	Function Call
Descriptio n	Means that a function call to another part of the system or class library is incorrect or missing.
Defect	Parameter
Descriptio n	Means that a function call or other interaction mechanism has an incorrect or missing parameter

9. Logic Defects

Defect	Compare
Descriptio n	means a mistake in a comparison statement
Notes	Examples: the wrong element is compared, a required comparison is missing, or the wrong type of comparison is made
Defect	Compute
Descriptio n	Such defects are made when computations produce incorrect results.
Defect	Wrong Location
Descriptio n	means that a correct operation is performed, but it is done too soon or too late.
Defect	Algorithm/Performance
Descriptio n	means that an inefficient algorithm is used.
Notes	Examples: performing unnecessary searches, passing large data arrays by value, and re-calculating values rather than storing them in variables
Defect	Other
Descriptio n	All other defect in this catogory
Notes	Examples: the need to use a loop instead of a single function call, missing an entire comparison statement and the required logic to handle the particular case, and using if-else blocks with two logically unrelated if blocks.

10. Larger Defects

Defect	Completeness
Descriptio n	A feature is partially implemented
Defect	GUI
Descriptio n	Defects in the user interface code relating to the consistency of the user-interface, and to the options made possible to the user in each situation.
Defect	Check outside code
Descriptio n	Defects that required that part of the application code that was not under review to be checked, as it was likely to contain incorrect code based on the current review.

B GLM – Precise Model Coefficients

For the calculation of the GLM, we used the R package `MASS` [VR02, R C13], and the following command:

```
glm.nb(formula = NumberOfTodos ~ CodeChurn + NumberOfChangedFiles +
     Tracker + MainBundle + Author, maxit = 500,
     init.theta = 0.4000689894, link = log)
```

The command calculated a model with the following parameter coefficients.

Coefficient	**Value**
Error Term	
(Intercept)	-34.004192
Code Churn	
CodeChurn	0.002574
Changed Files	
NumberOfChangedFiles	0.048253
Tracker	
corrective	-0.650777
adaptive	0.527672
preventive	-0.728885
perfective	0.701534
MainBundle	
edu.tum.cs.conqat.ada	0.296053
edu.tum.cs.conqat.architecture	0.476954
edu.tum.cs.conqat.cd_incubator	-0.937791
edu.tum.cs.conqat.clonedetective	0.324762
edu.tum.cs.conqat.commons	0.793104
edu.tum.cs.conqat.coverage	-36.786918
edu.tum.cs.conqat.cpp	-36.354702
edu.tum.cs.conqat.database	-0.577904
edu.tum.cs.conqat.dotnet	0.353873
edu.tum.cs.conqat.filesystem	0.298028
edu.tum.cs.conqat.findbugs	-35.107270
edu.tum.cs.conqat.graph	-35.941594
edu.tum.cs.conqat.html_presentation	0.167364
edu.tum.cs.conqat.io	-0.568088
edu.tum.cs.conqat.java	0.519543
edu.tum.cs.conqat.klocwork	-22.025370
edu.tum.cs.conqat.model_clones	-35.455045

edu.tum.cs.conqat.quamoco	1.961523
edu.tum.cs.conqat.regressiontest	0.200431
edu.tum.cs.conqat.scripting	-35.754787
edu.tum.cs.conqat.self	-0.150049
edu.tum.cs.conqat.simion	0.468814
edu.tum.cs.conqat.simulink	-0.960608
edu.tum.cs.conqat.sourcecode	0.343499
edu.tum.cs.conqat.svn	-36.106515
edu.tum.cs.conqat.systemtest	-35.532797
edu.tum.cs.conqat.text	1.146001
edu.tum.cs.conqat.tracking	-1.945797
org.conqat.android.metrics	-4.168671
org.conqat.engine.abap	1.167765
org.conqat.engine.api_analysis	0.907535
org.conqat.engine.architecture	0.638846
org.conqat.engine.blocklib	-36.948434
org.conqat.engine.bugzilla	-36.647132
org.conqat.engine.clone_tracking	2.257996
org.conqat.engine.code_clones	-0.079704
org.conqat.engine.codesearch	-0.518992
org.conqat.engine.commons	0.715151
org.conqat.engine.core	0.409059
org.conqat.engine.cpp	0.167496
org.conqat.engine.dotnet	0.614509
org.conqat.engine.graph	0.558142
org.conqat.engine.html_presentation	0.276341
org.conqat.engine.incubator	0.214965
org.conqat.engine.index	0.966182
org.conqat.engine.io	-1.978735
org.conqat.engine.java	0.801053
org.conqat.engine.levd	-5.550055
org.conqat.engine.persistence	1.003474
org.conqat.engine.report	0.808321
org.conqat.engine.repository	2.021809
org.conqat.engine.resource	0.213969
org.conqat.engine.self	-37.333548
org.conqat.engine.server	-37.621400
org.conqat.engine.service	-0.270366
org.conqat.engine.simulink	1.616546
org.conqat.engine.sourcecode	0.784824
org.conqat.engine.systemtest	1.870592
org.conqat.engine.text	0.133910
org.conqat.ide.architecture	-0.950501
org.conqat.ide.clones	0.075398
org.conqat.ide.commons.gef	-0.356652
org.conqat.ide.commons.ui	0.001236

org.conqat.ide.core	0.842194
org.conqat.ide.dev_tools	0.829869
org.conqat.ide.editor	0.179334
org.conqat.ide.findings	2.281378
org.conqat.ide.index.analysis	-4.333337
org.conqat.ide.index.core	-0.275343
org.conqat.ide.index.dev	-39.310811
org.conqat.lib.bugzilla	0.087937
org.conqat.lib.commons	0.947664
org.conqat.lib.parser	1.739108
org.conqat.lib.scanner	-1.234565
org.conqat.lib.simulink	2.565453
Author	
bader	34.631486
beller	35.562336
besenreu	-3.317127
deissenb	32.449625
feilkas	31.312504
goede	33.752369
heinemann	33.206064
herrmama	33.432585
hodaie	-3.935601
hummel	32.753639
juergens	32.796284
junkerm	31.903729
kanis	34.042702
kinnen	31.927869
klenkm	32.747542
lochmann	34.915264
ludwigm	-4.271351
malinskyi	NA
pfaller	33.255721
plachot	33.805882
poehlmann	33.535303
steidl	-3.021712
stemplinger	35.001225
streitel	34.719616
svejda	NA

```
Degrees of Freedom: 971 Total (i.e. Null);  867 Residual
Null Deviance:     1387

Deviance Residuals:
     Min        1Q    Median        3Q       Max
-2.45004  -0.99200  -0.61802   0.00664   2.50223
```

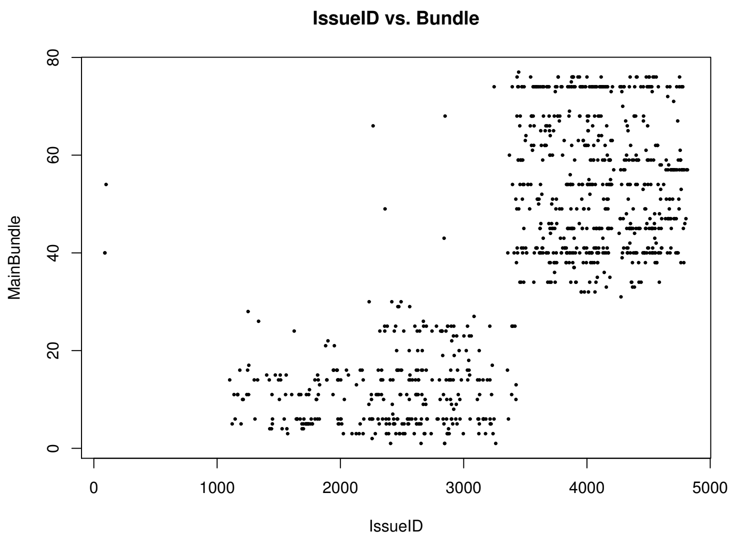


Figure B.1: The IssueID versus the lexicographically sorted MainBundle.

An interesting side-observation from the GLM in figure B.1 is the clearly-visible trunk move from `edu.tum.*` to `org.conqat.*` bundles. The move happened around issue 3200.

For the sake of completeness, we provide the raw output from R.

```
Coefficients: (2 not defined because of singularities)

                                              Estimate Std. Error z value  Pr(>|z|)
 (Intercept)                                -3.400e+01  1.383e+07   0.000  1.0000
 CodeChurn                                   2.574e-03  3.138e-04   8.203  2.34e-16 ***
 NumberOfChangedFiles                        4.825e-02  4.391e-03  10.989  < 2e-16 ***
 Trackercorrective                          -6.508e-01  3.315e-01  -1.963  0.0496 *
 Trackeradaptive                             5.277e-01  3.470e-01   1.521  0.1283
 Trackerpreventive                          -7.289e-01  4.584e-01  -1.590  0.1118
 Trackerperfective                           7.015e-01  2.850e-01   2.461  0.0138 *
 MainBundleedu.tum.cs.conqat.ada             2.961e-01  2.136e+00   0.139  0.8898
 MainBundleedu.tum.cs.conqat.architecture    4.770e-01  1.048e+00   0.455  0.6489
 MainBundleedu.tum.cs.conqat.cd_incubator   -9.378e-01  1.233e+00  -0.761  0.4469
 MainBundleedu.tum.cs.conqat.clonedetective  3.248e-01  1.001e+00   0.324  0.7456
 MainBundleedu.tum.cs.conqat.commons         7.931e-01  9.806e-01   0.809  0.4186
 MainBundleedu.tum.cs.conqat.coverage       -3.679e+01  6.711e+07   0.000  1.0000
 MainBundleedu.tum.cs.conqat.cpp            -3.635e+01  6.711e+07   0.000  1.0000
 MainBundleedu.tum.cs.conqat.database       -5.779e-01  1.326e+00  -0.436  0.6629
 MainBundleedu.tum.cs.conqat.dotnet          3.539e-01  1.075e+00   0.329  0.7420
 MainBundleedu.tum.cs.conqat.filesystem      2.980e-01  1.013e+00   0.294  0.7686
 MainBundleedu.tum.cs.conqat.findbugs       -3.511e+01  6.711e+07   0.000  1.0000
 MainBundleedu.tum.cs.conqat.graph          -3.594e+01  3.662e+07   0.000  1.0000
 MainBundleedu.tum.cs.conqat.html_presentation  1.674e-01  1.009e+00   0.166  0.8683
 MainBundleedu.tum.cs.conqat.io             -5.681e-01  1.247e+00  -0.456  0.6486
 MainBundleedu.tum.cs.conqat.java            5.195e-01  1.016e+00   0.512  0.6090
 MainBundleedu.tum.cs.conqat.klocwork       -2.203e+01  3.110e+00  -7.082  1.42e-12 ***
 MainBundleedu.tum.cs.conqat.model_clones   -3.546e+01  6.711e+07   0.000  1.0000
 MainBundleedu.tum.cs.conqat.quamoco         1.962e+00  1.504e+00   1.304  0.1922
 MainBundleedu.tum.cs.conqat.regressiontest  2.004e-01  1.224e+00   0.164  0.8700
 MainBundleedu.tum.cs.conqat.scripting      -3.575e+01  4.745e+07   0.000  1.0000
 MainBundleedu.tum.cs.conqat.self           -1.500e-01  1.618e+00  -0.093  0.9261
 MainBundleedu.tum.cs.conqat.simion          4.688e-01  1.300e+00   0.361  0.7183
 MainBundleedu.tum.cs.conqat.simulink       -9.606e-01  1.228e+00  -0.782  0.4340
 MainBundleedu.tum.cs.conqat.sourcecode      3.435e-01  1.049e+00   0.327  0.7434
 MainBundleedu.tum.cs.conqat.svn            -3.611e+01  4.745e+07   0.000  1.0000
 MainBundleedu.tum.cs.conqat.systemtest     -3.553e+01  6.711e+07   0.000  1.0000
 MainBundleedu.tum.cs.conqat.text            1.146e+00  1.385e+00   0.827  0.4080
 MainBundleedu.tum.cs.conqat.tracking       -1.946e+00  1.568e+00  -1.241  0.2147
 MainBundleorg.conqat.android.metrics       -4.169e+00  6.852e+07   0.000  1.0000
 MainBundleorg.conqat.engine.abap            1.168e+00  1.416e+00   0.825  0.4095
 MainBundleorg.conqat.engine.api_analysis    9.075e-01  1.414e+00   0.642  0.5211
 MainBundleorg.conqat.engine.architecture    6.388e-01  1.051e+00   0.608  0.5434
 MainBundleorg.conqat.engine.blocklib         -3.695e+01  4.745e+07   0.000  1.0000
 MainBundleorg.conqat.engine.bugzilla         -3.665e+01  4.745e+07   0.000  1.0000
 MainBundleorg.conqat.engine.clone_tracking    2.258e+00  1.478e+00   1.528  0.1265
 MainBundleorg.conqat.engine.code_clones      -7.970e-02  1.029e+00  -0.077  0.9383
 MainBundleorg.conqat.engine.codesearch       -5.190e-01  2.122e+00  -0.245  0.8067
 MainBundleorg.conqat.engine.commons           7.152e-01  9.787e-01   0.731  0.4649
 MainBundleorg.conqat.engine.core              4.091e-01  1.023e+00   0.400  0.6892
 MainBundleorg.conqat.engine.cpp               1.675e-01  1.590e+00   0.105  0.9161
 MainBundleorg.conqat.engine.dotnet            6.145e-01  1.255e+00   0.490  0.6243
 MainBundleorg.conqat.engine.graph             5.581e-01  1.458e+00   0.383  0.7018
 MainBundleorg.conqat.engine.html_presentation  2.763e-01  9.963e-01   0.277  0.7815
 MainBundleorg.conqat.engine.incubator         2.150e-01  1.107e+00   0.194  0.8460
 MainBundleorg.conqat.engine.index             9.662e-01  1.076e+00   0.898  0.3690
 MainBundleorg.conqat.engine.io               -1.979e+00  1.710e+00  -1.157  0.2472
 MainBundleorg.conqat.engine.java              8.011e-01  1.045e+00   0.767  0.4432
 MainBundleorg.conqat.engine.levd             -5.550e+00  2.183e+00  -2.542  0.0110 *
 MainBundleorg.conqat.engine.persistence       1.003e+00  1.044e+00   0.961  0.3365
 MainBundleorg.conqat.engine.report            8.083e-01  1.504e+00   0.537  0.5910
 MainBundleorg.conqat.engine.repository        2.022e+00  1.411e+00   1.433  0.1520
 MainBundleorg.conqat.engine.resource          2.140e-01  1.007e+00   0.212  0.8318
 MainBundleorg.conqat.engine.self             -3.733e+01  4.745e+07   0.000  1.0000
 MainBundleorg.conqat.engine.server           -3.762e+01  6.711e+07   0.000  1.0000
 MainBundleorg.conqat.engine.service          -2.704e-01  1.067e+00  -0.253  0.8000
 MainBundleorg.conqat.engine.simulink          1.617e+00  1.368e+00   1.182  0.2373
 MainBundleorg.conqat.engine.sourcecode        7.848e-01  1.008e+00   0.779  0.4360
 MainBundleorg.conqat.engine.systemtest        1.871e+00  1.260e+00   1.484  0.1377
 MainBundleorg.conqat.engine.text              1.339e-01  1.431e+00   0.094  0.9255
 MainBundleorg.conqat.ide.architecture        -9.505e-01  1.119e+00  -0.850  0.3955
 MainBundleorg.conqat.ide.clones               7.540e-02  1.166e+00   0.065  0.9484
 MainBundleorg.conqat.ide.commons.gef         -3.567e-01  1.332e+00  -0.268  0.7888
 MainBundleorg.conqat.ide.commons.ui           1.236e-03  1.195e+00   0.001  0.9992
 MainBundleorg.conqat.ide.core                 8.422e-01  1.115e+00   0.755  0.4500
 MainBundleorg.conqat.ide.dev_tools            8.299e-01  1.303e+00   0.637  0.5242
 MainBundleorg.conqat.ide.editor               1.793e-01  1.042e+00   0.172  0.8633
 MainBundleorg.conqat.ide.findings             2.281e+00  1.935e+00   1.179  0.2385
 MainBundleorg.conqat.ide.index.analysis      -4.333e+00  6.852e+07   0.000  1.0000
 MainBundleorg.conqat.ide.index.core          -2.753e-01  2.084e+00  -0.132  0.8949
 MainBundleorg.conqat.ide.index.dev           -3.931e+01  6.711e+07   0.000  1.0000
 MainBundleorg.conqat.lib.bugzilla             8.794e-02  1.466e+00   0.060  0.9522
 MainBundleorg.conqat.lib.commons              9.477e-01  9.808e-01   0.966  0.3339
 MainBundleorg.conqat.lib.parser               1.739e+00  1.879e+00   0.926  0.3546
 MainBundleorg.conqat.lib.scanner             -1.235e+00  1.098e+00  -1.124  0.2610
 MainBundleorg.conqat.lib.simulink             2.565e+00  2.013e+00   1.275  0.2024
 Authorbader                                   3.463e+01  1.383e+07   0.000  1.0000
 Authorbeller                                  3.556e+01  1.383e+07   0.000  1.0000
 Authorbesenreu                               -3.317e+00  6.852e+07   0.000  1.0000
 Authordeissenb                                3.245e+01  1.383e+07   0.000  1.0000
 Authorfeilkas                                 3.131e+01  1.383e+07   0.000  1.0000
 Authorgoede                                   3.375e+01  1.383e+07   0.000  1.0000
 Authorheinemann                               3.321e+01  1.383e+07   0.000  1.0000
 Authorherrmama                                3.343e+01  1.383e+07   0.000  1.0000
 Authorhodaie                                 -3.936e+00  6.852e+07   0.000  1.0000
 Authorhummel                                  3.275e+01  1.383e+07   0.000  1.0000
 Authorjuergens                                3.280e+01  1.383e+07   0.000  1.0000
 Authorjunkerm                                 3.190e+01  1.383e+07   0.000  1.0000
 Authorkanis                                   3.404e+01  1.383e+07   0.000  1.0000
 Authorkinnen                                  3.193e+01  1.383e+07   0.000  1.0000
 Authorklenkm                                  3.275e+01  1.383e+07   0.000  1.0000
 Authorlochmann                                3.492e+01  1.383e+07   0.000  1.0000
 Authorludwigm                                -4.271e+00  6.852e+07   0.000  1.0000
 Authormalinskyi                                      NA         NA      NA      NA
 Authorpfaller                                 3.326e+01  1.383e+07   0.000  1.0000
 Authorplachot                                 3.381e+01  1.383e+07   0.000  1.0000
 Authorpoehlmann                               3.354e+01  1.383e+07   0.000  1.0000
 Authorsteidl                                 -3.022e+00  4.943e+07   0.000  1.0000
 Authorstemplinger                             3.500e+01  1.383e+07   0.000  1.0000
 Authorstreitel                                3.472e+01  1.383e+07   0.000  1.0000
 Authorsvejda                                         NA         NA      NA      NA
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for Negative Binomial(0.4001) family taken to be 1)

    Null deviance: 1387.25  on 971  degrees of freedom
Residual deviance:  763.27  on 867  degrees of freedom
AIC: 3205

Number of Fisher Scoring iterations: 1


              Theta:  0.4001
          Std. Err.:  0.0317

 2 x log-likelihood:  -2993.0240
```

Bibliography

[53910] IEEE standard classification for software anomalies. *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, pages 1–23, 2010.

[Ada84] J. Adair. The hawthorne effect: A reconsideration of the methodological artifact. *Journal of applied psychology*, 69(2):334, 1984.

[AGDS07] E. Arisholm, H. Gallis, T. Dyba, and D. Sjoberg. Evaluating pair programming with respect to system complexity and programmer expertise. *Software Engineering, IEEE Transactions on*, 33(2):65–86, 2007.

[AL06] B. Abraham and J. Ledolter. *Introduction to regression modeling*. Thomson Brooks/Cole, 2006.

[BA04] K. Beck and C. Andres. *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2004.

[Bak97] R. Baker. Code reviews enhance software quality. In *Proceedings of the 19th international conference on Software engineering*, pages 570–571. ACM, 1997.

[BB05] 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.

[BB13] A. Bacchelli and Ch. 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.

[BLV01] A. Bianchi, F. Lanubile, and G. Visaggio. A controlled experiment to assess the effectiveness of inspection meetings. In *Software Metrics Symposium, 2001. METRICS 2001. Proceedings. Seventh International*, pages 42–50. IEEE, 2001.

[BMG10] M. Bernhart, A. Mauczka, and T. Grechenig. Adopting code reviews for agile software development. In *Agile Conference (AGILE), 2010*, pages 44–47. IEEE, 2010.

[Bro87] F. Brooks. No silver bullet-essence and accidents of software engineering. *IEEE computer*, 20(4):10–19, 1987.

[Bug] Bugzilla. `http://www.bugzilla.org/`. Accessed 2013/10/13.

[BvdSvD95] H. Berendsen, D. van der Spoel, and R. van Drunen. GROMACS: A message-passing parallel molecular dynamics implementation. *Computer Physics Communications*, 91(1):43–56, 1995.

[BZ05] V. Berger and J. Zhang. *Structural Zeros*. John Wiley & Sons, Ltd, 2005.

[CBC$^+$92] 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 Trans. Software Eng.*, 18(11):943–956, 1992.

[CdSH$^+$03] L. Cheng, C. de Souza, S. Hupfer, J. Patterson, and S. Ross. Building collaboration into ides. *Queue*, 1(9):40, 2003.

[CLR$^+$02] M. Ciolkowski, O. Laitenberger, D. Rombach, F. Shull, and D. Perry. Software inspections, reviews and walkthroughs. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 641–642. IEEE, 2002.

[CMKC03] M. Cusumano, A. MacCormack, Ch. Kemerer, and B. Crandall. Software development worldwide: The state of the practice. *Software, IEEE*, 20(6):28–34, 2003.

[Coh60] J. Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

[Con] ConQAT. `http://www.conqat.org`. Accessed 2013/08/30.

[CW00] A. Cockburn and L. Williams. The costs and benefits of pair programming. *Extreme programming examined*, pages 223–247, 2000.

[Dei09] F. Deißenböck. *Continuous Quality Control of Long-Lived Software Systems*. PhD thesis, 2009.

[DHJS11] Deißenböck, F., U. Hermann, E. Jürgens, and T. Seifert. LEvD: A lean evolution and development process. `https://conqat.cqse.eu/download/levd-process.pdf`, 2011. Accessed 2013/08/30.

[DM03] J. Duraes and H. Madeira. Definition of software fault emulation operators: A field data study. In *Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on*, pages 105–114. IEEE, 2003.

[Ent] GitHub Enterprise. `https://enterprise.github.com/`. Accessed 2013/10/14.

[EPSK01] Ch. Ebert, C. Parro, R. Suttels, and H. Kolarczyk. Improving validation activities in a global software development. In *Proceedings of the 23rd international Conference on Software Engineering*, pages 545–554. IEEE Computer Society, 2001.

[EW98] K. El Emam and I. Wieczorek. The repeatability of code defect classifications. In *Software Reliability Engineering, 1998. Proceedings. The Ninth International Symposium on*, pages 322–333. IEEE, 1998.

[Fag76] M. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[Fah] K. Fahrmeir. Regression, modelle, methoden und anwendungen.

[Fin] FindBugs™. http://findbugs.sourceforge.net/. Accessed 2013/08/30.

[Fle81] J. Fleiss. Statistical methods for rates and proportions, 1981.

[FxC] FxCop. http://findbugs.sourceforge.net/. Accessed 2013/08/30.

[Ger] Gerrit. https://code.google.com/p/gerrit/. Accessed 2013/10/09.

[GHJV93] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Abstraction and reuse of object-oriented design*. Springer, 1993.

[Gita] Git. http://git-scm.com/.

[Gitb] GitHub. https://github.com/. Accessed 2013/10/09.

[Gmb] CQSE GmbH. http://www.cqse.eu. Accessed 2013/10/14.

[GN96] P. Greenwood and M. Nikulin. *A guide to chi-squared testing*, volume 280. Wiley-Interscience, 1996.

[Gra92] R. Grady. *Practical software metrics for project management and process improvement*, volume 3. Prentice Hall Englewood Cliffs, 1992.

[Gre94] W. Greene. Accounting for excess zeros and sample selection in poisson and negative binomial regression models. 1994.

[Gro] Gromacs. http://www.gromacs.org. Accessed 2013/09/02.

[GW06] T. Gorschek and C. Wohlin. Requirements abstraction model. *Requirements Engineering*, 11(1):79–101, 2006.

[Hat08] L. Hatton. Testing the value of checklists in code inspections. *Software, IEEE*, 25(4):82–88, 2008.

[HR90] J. Hartmann and D. Robson. Techniques for selective revalidation. *Software, IEEE*, 7(1):31–36, 1990.

[Hum95] W. Humphrey. A discipline for software engineering. 1995.

[Jen] Jenkins. http://jenkins-ci.org/. Accessed 2013/10/14.

[Jir] Jira. https://www.atlassian.com/de/software/jira. Accessed 2013/10/13.

[Ken06] N. Kennedy. Google Mondrian: web-based code review and storage. http://www.niallkennedy.com/blog/2006/11/google-mondrian.html, 2006. Accessed 2013/10/14.

[KK09] S. Kollanus and J. Koskinen. Survey of software inspection research. *The Open Software Engineering Journal*, 3(1):15–34, 2009.

[KM93] J. Knight and E Myers. An improved inspection technique. *Communications of the ACM*, 36(11):51–61, 1993.

[KP09a] C. Kemerer and M. Paulk. The impact of design and code reviews on software quality: An empirical study based on psp data. *Software Engineering, IEEE Transactions on*, 35(4):534–550, 2009.

[KP09b] Ch. Kemerer and M. Paulk. The Impact of Design and Code Reviews on Software Quality: An Empirical Study Based on PSP Data. *IEEE Trans. Software Eng.*, 35(4):534–550, 2009.

[KPHR02] B. Kitchenham, S. Pfleeger, D. Hoaglin, and J. Rosenberg. Preliminary Guidelines for Empirical Research in Software Engineering. *IEEE Trans. Software Engineering*, 28(8):721 – 734, August 2002.

[Kre99] Ch. Krebs. *Ecological methodology*, volume 620. Benjamin/Cummings Menlo Park, California, 1999.

[Lau08] A. Laurent. *Understanding open source and free software licensing*. O'Reilly, 2008.

[Lib] LibreOffice. `http://www.libreoffice.org/`. Accessed 2013/09/10.

[Mar03] R. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[MDL87] H. Mills, M. Dyer, and R. Linger. Cleanroom software engineering. 1987.

[Mer] Mercurial. `http://mercurial.selenic.com/`. Accessed 2013/10/13.

[Mey08] B. Meyer. Design and code reviews in the age of the internet. *Communications of the ACM*, 51(9):66–71, 2008.

[Mil13] L. Milanesio. *Learning Gerrit Code Review*. Packt Publishing Ltd, 2013.

[ML09] M. Mäntylä and C. Lassenius. What Types of Defects Are Really Discovered in Code Reviews? *IEEE Trans. Software Eng.*, 35(3):430–448, 2009.

[MRZ$^+$05] J. Maranzano, S. Rozsypal, G. Zimmerman, G. Warnken, P. Wirth, and D. Weiss. Architecture reviews: Practice and experience. *Software, IEEE*, 22(2):34–43, 2005.

[Mül04] M. Müller. Are reviews an alternative to pair programming? *Empirical Software Engineering*, 9(4):335–351, 2004.

[Mül05] M. Müller. Two controlled experiments concerning the comparison of pair programming to peer review. *Journal of Systems and Software*, 78(2):166–179, 2005.

[MWR98] J. Miller, M. Wood, and M. Roper. Further experiences with scenarios and checklists. *Empirical Software Engineering*, 3(1):37–64, 1998.

[Mye86] E. Myers. Ano (nd) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

[Ohl] Ohloh.net. `http://www.ohloh.net/p/gromacs`. Accessed 2013/09/02.

[Per] Perforce. `http://www.perforce.com/`. Accessed 2013/10/13.

[Pha] Phabricator. `http://phabricator.org/`. Accessed 2013/10/14.

[PlaBC] Plato. *Gorgias*. 390/387 B.C.

[PMD] PMD. `http://pmd.sourceforge.net/`. Accessed 2013/08/30.

[Pro] The Trac Project. `http://trac.edgewall.org/`. Accessed 2013/10/13.

[PV94] A. Porter and L. Votta. An experiment to assess different defect detection methods for software requirements inspections. In *Proceedings of the 16th international conference on Software engineering*, pages 103–112. IEEE Computer Society Press, 1994.

[R C13] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013.

[RA00] T. Ritzau and J. Andersson. Dynamic deployment of java applications. In *Java for Embedded Systems Workshop*, volume 1, page 21. Citeseer, 2000.

[RAT$^+$06] P. Runeson, C. Andersson, T. Thelin, A. Andrews, and T. Berling. What do we know about defect detection methods? *Software, IEEE*, 23(3):82–90, 2006.

[Red] Redmine. `http://www.redmine.org/`. Accessed 2013/10/13.

[Rev] Mylyn Reviews. `http://www.eclipse.org/reviews/`. Accessed 2013/08/30.

[SHJ13] D. Steidl, B. Hummel, and E. Jürgens. Quality analysis of source code comments. In *Proceedings of the 21st IEEE Internation Conference on Program Comprehension (ICPC'13)*, 2013.

[SJLY00] Ch. Sauer, R. Jeffery, L. Land, and Ph. Yetton. The effectiveness of software development technical reviews: A behaviorally motivated program of research. *Software Engineering, IEEE Transactions on*, 26(1):1–14, 2000.

[SKI04] G. Sabaliauskaite, Sh. 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.

[Sty] StyleCop. `http://stylecop.codeplex.com/`. Accessed 2013/08/30.

[Sub] Apache™ Subversion®. `http://subversion.apache.org/`. Accessed 2013/10/13.

[SV01] H. Siy and L. Votta. Does the Modern Code Inspection Have Value? In *ICSM*, page 281, 2001.

[Sys] Concurrent Versions System. `http://savannah.nongnu.org/projects/cvs`. Accessed 2013/10/14.

[Tea] Teamscale. `http://www.teamscale.org`. Accessed 2013/09/11.

[UNMM06] 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.

[VG05] A. Viera and J. Garrett. Understanding interobserver agreement: the kappa statistic. *Fam Med*, 37(5):360–363, 2005.

[Vot93] L. Votta. Does every inspection need a meeting? In *ACM SIGSOFT Software Engineering Notes*, volume 18, pages 107–114. ACM, 1993.

[VR02] W. Venables and B. Ripley. *Modern Applied Statistics with S*. Springer, New York, fourth edition, 2002. ISBN 0-387-95457-0.

[Wag08] 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.

[WF84] G. Weinberg and D. Freedman. Reviews, walkthroughs, and inspections. *Software Engineering, IEEE Transactions on*, (1):68–72, 1984.

[WG68] M. Wilk and R. Gnanadesikan. Probability plotting methods for the analysis for the analysis of data. *Biometrika*, 55(1):1–17, 1968.

[WH11] C. Wu and M. Hamada. *Experiments: planning, analysis, and optimization*, volume 552. John Wiley & Sons, 2011.

[WJKT05] 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.

[WKCJ00] L. Williams, R. Kessler, W. Cunningham, and R. Jeffries. Strengthening the case for pair programming. *Software, IEEE*, 17(4):19–25, 2000.

[WRBM97a] M. Wood, M. Roper, A. Brooks, and J. Miller. Comparing and combining software defect detection techniques: a replicated empirical study. In *ACM SIGSOFT Software Engineering Notes*, volume 22, pages 262–277. Springer-Verlag New York, Inc., 1997.

[WRBM97b] M. Wood, M. Roper, A. Brooks, and J. Miller. Comparing and Combining Software Defect Detection Techniques: A Replicated Empirical Study. In *5th ACM SIGSOFT – Symposium on the Foundations of Software Engineering*, pages 262–277, September 1997.

[WYCL08] 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.