Cognitive-Support Code Review Tools

Improved Efficiency of Change-Based Code Review by Guiding and Assisting Reviewers

Von der Fakultät für Elektrotechnik und Informatik

der Gottfried Wilhelm Leibniz Universität Hannover

zur Erlangung des akademischen Grades

Doktor-Ingenieur

(abgekürzt: Dr.-Ing.)

genehmigte Dissertation von Herrn

M. Eng. Tobias Baum

geboren am 28.04.1985 in Hannover, Deutschland

2019

Betreuer: Prof. Dr. Kurt Schneider
1. Referent: Prof. Dr. Alberto Bacchelli
2. Referent: Prof. Dr. Sören Auer
Vorsitzende der Prüfungskommission: Prof. Dr. Eirini Ntoutsi

Tag der Promotion: 25. Oktober 2019

To my family . . .

Abstract

Code reviews, i.e., systematic manual checks of program source code by other developers, have been an integral part of the quality assurance canon in software engineering since their formalization by Michael Fagan in the 1970s. Computer-aided tools supporting the review process have been known for decades and are now widely used in software development practice. Despite this long history and widespread use, current tools hardly go beyond simple automation of routine tasks. The core objective of this thesis is to systematically develop options for improved tool support for code reviews and to evaluate them in the interplay of research and practice.

The starting point of the considerations is a comprehensive analysis of the state of research and practice. Interview and survey data collected in this thesis show that review processes in practice are now largely *change-based*, i.e., based on checking the changes resulting from the iterative-incremental evolution of software. This is true not only for open source projects and large technology companies, as shown in previous research, but across the industry. Despite the common change-based core process, there are various differences in the details of the review processes. The thesis shows possible factors influencing these differences. Important factors seem to be the process variants supported and promoted by the used review tool. In contrast, the used tool has little influence on the fundamental decision to use regular code reviews. Instead, the interviews and survey data suggest that the decision to use code reviews depends more on cultural factors.

Overall, the analysis of the state of research and practice shows that there is a potential for developing better code review tools, and this potential is associated with the opportunity to increase efficiency in software development. The present thesis argues that the most promising approach for better review support is reducing the reviewer's cognitive load when reviewing large code changes. Results of a controlled experiment support this reasoning. The thesis explores various possibilities for cognitive support, two of these in detail: Guiding the reviewer by identifying and presenting a good order of reading the code changes being reviewed, and assisting the reviewer through automatic determination of change parts that are irrelevant for review. In both cases, empirical data is used to both generate and test hypotheses. In order to demonstrate the practical suitability of the techniques, they are also used in a partner company in regular development practice. For this evaluation of the cognitive support techniques in practice, a review tool which is suitable for use in the partner company and as a platform for review research is needed. As such a tool was not available, the code review tool "CoRT" has been developed. Here, too, a combination of an analysis of the state of research, support of design decisions through scientific studies and evaluation in practical use was employed.

Overall, the results of this thesis can be roughly divided into three blocks: Researchers and practitioners working on improving review tools receive an empirically and theoretically sound catalog of requirements for cognitive-support review tools. It is available explicitly in the form of essential requirements and possible forms of realization, and additionally implicitly in the form

of the tool "CoRT". The second block consists of contributions to the fundamentals of review research, ranging from the comprehensive analysis of review processes in practice to the analysis of the impact of cognitive abilities (specifically, working memory capacity) on review performance. As the third block, innovative methodological approaches have been developed within this thesis, e.g., the use of process simulation for the development of heuristics for development teams and new approaches in repository and data mining.

Zusammenfassung

Code Reviews, d. h. manuelle Prüfungen von Programmquellcode durch andere Entwickler, sind seit ihrer Formalisierung durch Michael Fagan in den 1970er Jahren fester Bestandteil des Qualitätssicherungs-Kanons im Software Engineering. Auch computerbasierte Werkzeuge zur Unterstützung des Review-Prozesses sind seit Jahrzehnten bekannt und inzwischen in der Softwareenwicklungspraxis weit verbreitet. Trotz dieser langen Historie und weiten Verbreitung gehen auch aktuelle Werkzeuge kaum über einfache Automatisierung anfallender Routineaufgaben hinaus. Das Kernziel der vorliegenden Arbeit ist, systematisch Möglichkeiten zur verbesserten Werkzeugunterstützung für Code Reviews zu erarbeiten und im Zusammenspiel von Wissenschaft und Praxis zu evaluieren.

Ausgangspunkt der Überlegungen ist eine umfassende Analyse des Standes der Forschung und der Praxis. Im Rahmen dieser Arbeit erhobene Interview- und Umfragedaten zeigen, dass Review-Prozesse in der Praxis inzwischen zu einem großen Teil *änderungsbasiert* sind, d.h. auf dem Prüfen von Änderungen, die sich aus der iterativ-inkrementellen Weiterentwicklung von Software ergeben, basieren. Dies gilt nicht nur, wie in der vorherigen Forschung gezeigt wurde, für Open-Source-Projekte und große Technologiefirmen, sondern in der gesamten Branche. Dem gemeinsamen änderungsbasierten Kernprozess stehen jedoch diverse Unterschiede in der Detailumsetzung der Reviewprozesse gegenüber. Die Arbeit zeigt mögliche Einflussfaktoren für diese Unterschiede auf. Wichtige Faktoren scheinen die vom genutzen Review-Werkzeug unterstützten und bevorzugten Prozessvarianten zu sein. Diesem großen Einfluss der verwendeten Werkzeuge auf die Ausprägung des Review-Prozesses steht ein geringer Einfluss auf die grundsätzliche Entscheidung für die Nutzung regelmäßiger Code Reviews entgegen. Die Interviews und Umfragedaten legen stattdessen nahe, dass die Entscheidung für die Nutzung von Code Reviews eher von kulturellen Faktoren abhängig ist.

Aus der Analyse des Stands der Forschung und der Praxis ergibt sich insgesamt, dass es ein Potential zur Entwicklung besserer Code-Review-Werkzeuge gibt und mit diesem Potential die Chance zur Effizienzsteigerung in der Softwareentwicklung einhergeht. In der vorliegenden Arbeit wird dargelegt, dass der vielversprechendste Ansatz für bessere Review-Unterstützung in der kognitiven Entlastung des Reviewers beim Prüfen großer Code-Änderungen besteht. Diese Argumentation kann durch Ergebnisse eines kontrollierten Experiments gestützt werden. Die Arbeit untersucht verschiedene Möglichkeiten zur kognitiven Unterstützung, zwei davon im Detail: Das Leiten des Reviewers durch Ermitteln und Präsentieren einer guten Abfolge des Lesens der zu prüfenden Code-Änderungen sowie das Entlasten des Reviewers durch automatisches Erkennen für das Review irrelevanter Teile der Änderungen. In beiden Fällen werden empirische Daten sowohl zur Generierung als auch zur Prüfung von Hypothesen verwendet. Um die Praxistauglichkeit der Unterstützungsvarianten zu zeigen, werden diese zusätzlich in einem Partnerunternehmen in der regulären Entwicklungspraxis eingesetzt. Für diese Evaluation der kognitiven Unterstützungsfunktionen im Praxiseinsatz wird ein Review-Werkzeug benötigt, das sich für den

Einsatz im Partnerunternehmen und als Plattform für Reviewforschung eignet. Da ein solches Werkzeug nicht verfügbar war, wurde im Rahmen dieser Arbeit das Code-Review-Werkzeug „CoRT" entwickelt. Auch hierbei wurde wieder auf eine Kombination aus einer Analyse des Standes der Forschung, Unterstützung von Designentscheidungen durch wissenschaftliche Studien und Evaluation im Praxiseinsatz zurückgegriffen.

Insgesamt lassen sich die Ergebnisse dieser Arbeit grob in drei Blöcke einteilen: Forscher und Praktiker, die an der Verbesserung von Review-Werkzeugen arbeiten, erhalten einen empirisch und theoretisch fundierten Katalog von Anforderungen an kognitiv-unterstützende Review-Werkzeuge. Dieser liegt einmal explizit in Form essentieller Anforderungen und möglicher Realisierungsformen vor, und zusätzlich implizit in Form der im Praxiseinsatz bewährten Ausprägung „CoRT". Der zweite Block besteht aus Beiträgen zu den Grundlagen der Reviewforschung und erstreckt sich von der umfassenden Analyse von Review-Prozessen in der Praxis bis hin zur Analyse des Einflusses kognitiver Fähigkeiten (konkret der Arbeitsgedächtniskapazität) auf die Reviewleistung. Aus den erzielten Ergebnissen ergeben sich Ansätze für weitere Forschung und die weitere Verbesserung der Softwareentwicklung durch Code Reviews. Als dritter Block wurden im Rahmen dieser Arbeit innovative methodische Ansätze ausgearbeitet, u.a. die Nutzung von Prozesssimulation zur Erarbeitung von Heuristiken für Entwicklungsteams und neue Ansätze im Repository- und Data-Mining.

Schlagworte: Software Engineering, Code Reviews, kognitive Last, Quellcode-Lesetechniken, Software-Entwicklungswerkzeuge

Acknowledgments

While working on this thesis, I had the luck to meet and work with a large number of friendly and gifted people. Firstly, I would like to thank my supervisor, Kurt Schneider, for giving me the freedom to follow my interests and supporting me throughout the process. Further major thanks need to go to Alberto Bacchelli, who not only was a great collaborator on several of my studies but also introduced me to other bright people from his Delft und Zürich groups.

As another great collaborator, Steffen Herbold provided valuable input on the data mining study at the end of the thesis. In Hannover, I had the chance to work with and learn from several of the other Ph.D. students and postdocs. Leif Singer and Raphael Pham provided valuable advice on my Grounded Theory studies, Olga Boruszweski and Kai Niklas were great collaborators in my early studies and Fabian Kortum, Katja Fuhrmann and Javad Ghofrani helped later on. Perhaps even more importantly, they and many of the other colleagues helped to create a positive atmosphere and became friends over time. Beyond the software engineering group, people from other groups and departments in Hannover were open to my questions when I needed them, most notably Matthias Becker on simulation and Eirini Ntoutsi on data mining. Besides, several students contributed to my research to varying degrees: Hendrik Lessmann, Roman Gripp, Martin Liro, Hakan Sahin, and Rajib Das in Hannover, and Enrico Fregnan and Lorenzo Gasparini in Delft.

Without the teachers during my earlier studies at FHDW Hannover, especially Michael Löwe, Carsten Elsner, Harald König and Volkhard Klinger, I would not be where I am today.

It should not be taken for granted that a company supports its employees as much in pursuing their academic goals as my employer, SET GmbH, did. Therefore, I am greatly indebted to its founder, Till Dammermann, for making this possible. Many of my colleagues at SET had to deal with me being less available for a long time, and I want to thank them all. Special thanks go to Arthur Brack and Christoph Schulz for their contributions.

I am happy to have several good and long friends who also supported me during my thesis. Of them, Matthias Becker (Uni Bonn) deserves a special mention. His advice from his own Ph.D. experiences was invaluable, and his proof-reading skills helped to shape the thesis for the better. Final thanks need to go to my family. This thesis is for you.

Contents

Part IV   Conclusion

Part V   Appendix

1

Introduction

1.1 Motivation and Background

Software Engineering is often more about humans than about computers: Humans pose the requirements, humans collaborate to implement these requirements in software, and human characteristics lead to mistakes and inefficiencies in that process. This thesis, too, is about a software engineering technique in which humans play a major role: Code Reviews. In code reviews, software developers read each other's source code to find errors and other deficiencies and to learn and spread knowledge (see Chapter 4 for a more detailed definition). In many areas, humans use tools to increase their abilities, and the central hypothesis for this thesis is that the performance of human reviewers can be improved by providing better computer-based tools to help them during the review.

This hypothesis arose out of the author's experiences with current code review tools and with code review practices in a medium-sized software company. As is detailed in Chapter 7, review tools so far provide little more than support for book-keeping and data-handling. According to results from Microsoft as well as open-source projects, developers spend 10 to 15% of their time in code reviews [60]. This points to a considerable potential for cost savings by increasing review efficiency, not to mention savings through better quality or spreading of knowledge. Like the cited study, most current research on code reviews is based on open-source projects or large software companies. The emphasis of this thesis is different: Its scope is limited to commercial software development, and it takes care to include companies of different sizes in its survey of the state of the practice. When developing better code review support, this emphasis is amplified by applying the results in a medium-sized software company.

The analysis of the state of the art and practice shows possibilities for better review support. However, there is a vast number of possibilities, and to keep the thesis scope manageable not all of them can be studied. This thesis argues that cognitive support for the reviewer is the most promising route, and selects two possibilities for such cognitive support to study in detail.

All in all, these arguments lead to the following goal:

> The goal of this thesis is to analyze the state of the practice regarding the use of code reviews, to derive how code review tools should be improved to help software development teams most, and to study selected improvement possibilities in detail. The research is done with a focus on commercial software development, which is emphasized by applying the results in a medium-sized software company.

1.2 Contributions

This section lists the main contributions of the thesis and also some limitations. The thesis contains several contributions to research on improved code review tools and to general code review research, among those:

- The systematically derived notion of "cognitive-support code review tools", and an empirically and theoretically founded catalog of essential requirements for cognitive-support code review tools.
- An open-source implementation of a cognitive-support code review tool that showed its fitness for production use in a deployment in a software company.
- An empirically derived and tested theory on how to order code changes for optimal understanding in code reviews, and an implementation of this theory.
- An approach that uses repository mining to classify code change parts by their importance for review, tested in a company case study.
- An analysis of the state of the practice for code reviews, and grounded hypotheses on the use and non-use of reviews in general and on the use of certain review variants.
- An analysis of the effects and contextual factors that influence whether pre-commit or post-commit reviews lead to better results regarding efficiency, quality or cycle time.
- Evidence that the reviewer's working memory capacity is associated with code review effectiveness for certain kinds of defects.

Besides these contributions, the thesis also advances several Software Engineering research methods, for example with an approach to use software process simulation to derive heuristics for use in practice, and a data mining algorithm and system that is geared towards iterative feedback from domain experts.

One of the major limitations is the low statistical power for some conclusions from the controlled experiments. Most of the studied techniques were also deployed in commercial practice for triangulation, but this does not allow definite conclusions either, because there is no control group and it is hard to measure the effect on review outcomes. A second limitation is that the hypothesis of cognitive-support as the most promising avenue for improvement in code reviews is not compared to competing hypotheses with a controlled experiment, but only assessed by accumulating confirmatory evidence.

1.3 Approach and Research Methods

The general approach taken by this thesis is rooted in Hevner's three-cycle model of design science research [173], as depicted in Figure 1.1. Hevner's model emphasizes that research that creates artifacts to reach some human goal needs to be integrated with both, the application domain (relevance) and the scientific and domain body of knowledge (rigor). These integrations,
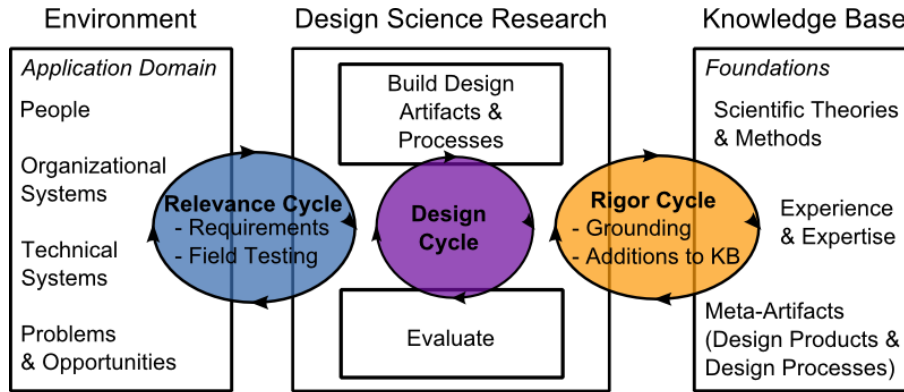
Figure 1.1: Hevner's three-cycle view of design science research, which is used as a methodological guide for this thesis. (based on [173])

as well as the creation of the artifact itself, are iterative, going back and forth between evaluation and incorporation of gathered knowledge. The central created artifact of this thesis is the code review tool *CoRT*, but the three-cycle view also influenced the creation of secondary artifacts, like the code ordering theory of Chapter 14 or the data mining system used in Chapter 15.

For the specific (sub-)studies to establish relevance and rigor, the research methods are chosen pragmatically based on the respective study goals and research questions. Many studies use a mixed-methods approach and triangulate several data sources. In sum, this leads to the use of a wide variety of different methods, like qualitative interviews and Grounded Theory hypothesis generation, controlled experiments, software process simulation, repository mining case studies, and semi-systematic literature reviews. The specific methods are described together with the respective studies.

1.4 Structure

The results are described linearly in the thesis, although an iterative research methodology has been used. The thesis is structured into three parts. Part I discusses background information on code reviews from the related work. In addition, it contains the results from three studies on code review use in commercial practice. That first part ensures that the remaining parts of this thesis are rooted in an in-depth analysis of current code review processes and their problems. Part II discusses the code review tool that is used to test some of the thesis' hypotheses in a company setting. It also describes a simulation study and a controlled experiment that were performed to inform design decisions for that code review tool. Part III finally revolves around cognitive-support code review tools. It first motivates, based on the results from the earlier parts, why cognitive-support is a sensible next step in the evolution of code review tools. Then, it describes two ideas for cognitive-support in detail: Showing the parts of the code change under review in an improved order, and using empirical data from previous reviews to classify parts that can be left out from the review. The final Part IV concludes the thesis and provides an outlook on future research opportunities.

Figure 1.2 depicts how findings and arguments from specific chapters are used in others. The thesis uses a variety of data sources, and many of them were reused in several chapters. Therefore, the figure also shows which data sources were used in which chapters.

Data Sources

- Semi-systematic literature review: Review Use
- Interviews on review use (24 participants, 19 companies)
- Online survey on review use (240 teams)
- Semi-systematic literature review: Review Tools
- Case company feedback on deployed tool
- Collaboration with two practitioners
- Survey in Java User Group (12 participants)
- Case company review statistics
- Controlled experiment (32 participants)
- Task-guided interviews on ordering (12 participants)
- Online survey on ordering (201 participants)
- Surveys in case company
- Controlled experiment (50 participants)
- SCM repository mining in company
- Feedback sessions in company

Thesis Chapters

Part I: Code Review in Industry - Why improved review support is worthwhile
2. Related Work on the State and History of Code Reviews in Industrial Practice
3. Methodology to Assess the State of the Practice
4. Reviews in Current Industrial Practice
5. Use and Non-Use of Reviews - Culture Beats More Efficient Tools
6. Variations in Industrial Review Processes
7. Tools and Techniques to Support Reviews

Part II The "CoRT" Code Review Tool and Research Platform
8. Context for Action Research on Improved Code Review Tooling: The Case Company
9. The Code Review Tool "CoRT"
10. A Simulation-Based Comparison of Pre- and Post-Commit Reviews
11. An Empirical Comparison of Multiple Ways to Present Source Code Diffs

Part III Cognitive Support: Guiding and Assisting Reviewers
12. Cognitive-Support Code Review Tools
13. An Experiment on Cognitive Load in Code Reviews
14. Ordering of Change Parts
15. Classification of Change Parts

Selected Findings

- Recent information on reviews in SMEs missing
- Teams use reviews to reach a combination of goals.
- Change-based review dominates in practice.
- Code review is most likely to remain in use if it is embedded into the process or tools so that it does not require a conscious decision to do a review.
- Better review tools will mainly help teams that already use reviews.
- Industrial review processes differ in various details, influenced by contextual factors and the used tool.
- Current tools are lacking advanced reviewer support
- Colored, aligned presentation of diffs is preferred by a majority of developers.
- In most industrial use cases, there is no striking benefit of pre-commit reviews.
- Reducing the cognitive load of reviewers will lead to better review performance. A new generation of review tools should support the reviewer when understanding large changes.
- A good order of reading code change parts can be formally defined and efficiently computed, but strong empirical support for its benefits is still lacking.
- Simple rules can reduce the size of changes that need to be reviewed. To find even more efficient rules with repository mining, several obstacles need to be overcome.

Figure 1.2: Connections between data sources, thesis chapters, and the flow of arguments and central findings. The colored bars on data sources denote the chapters they are used in. Arrows denote argument-flow; arrows that directly connect chapters denote motivating arguments. For literature studies, only (semi-)systematic studies are shown.

Part I

Code Review in Industry – Why improved review support is worthwhile

2

Related Work on the State and History of Code Reviews in Industrial Practice

Before starting out to work on improved code review tooling, the current state of the practice has to be known. Furthermore, the extent of the potential practical impact of work on review tooling should be checked. Therefore, the first part of this thesis describes the current state of the practice and motivates the search for better code review tools. It also provides information on the state and history of code review tools and related techniques.

This chapter surveys the existing research literature on code reviews in industrial practice. Before that, the topic is introduced by a brief overview of the history of code reviews and inspections. This shows the state of the art regarding the studies described in the other chapters of Part I of this thesis. The relevant state of the art for Parts II and III also contains research on review support and code review tools. Their description depends on the results brought forth in the following chapters. Therefore, an overview of related work on tools and techniques to support code reviews is postponed to Chapter 7, and further related work for cognitive-support is discussed in Chapter 12. Furthermore, several other chapters discuss related work that is specific to their respective contents.

Reduced to its core, code review is the proof-reading of program source code by one or several peers, and as such it is probably as old as programming itself. The first major scientific contribution appeared in 1976: Michael Fagan [117] had systematically compared various ways of reviewing code and other development artifacts in projects at IBM and developed a structured process which he called 'Inspection'. An Inspection is performed by a disciplined team of several people with distinct roles. In a Fagan Inspection, there is a phase of individual preparation and a review meeting in which the artifact is checked by the team. Fagan stressed the importance of the meeting synergy for finding defects. He noted that Inspections can be performed for all kinds of development artifacts, and that reviewing early artifacts like requirements and design documents is more important than reviewing late documents like code and tests. Following Fagan's publication, several other authors published variants of inspection processes (e.g., [140, 397]), and in 1988 an IEEE standard [179] was published that defines several review processes. Classification schemes for these processes have been published, among others, by Laitenberger and DeBaud [216] and Kim, Sauer and Jeffery [195]. Macdonald and Miller [238] even devel-

oped a domain-specific language for the description of inspection processes based on a detailed comparison of different processes from the literature.

Many aspects of inspections have been studied empirically: Several case studies indicate that for finding an optimal number of defects, one has to review slowly enough (i.e., with a low 'inspection rate') [218, 354]. A series of experiments (e.g., [187, 390]) found only small meeting gains, which raises doubts whether they are worth the scheduling effort. More reviewers usually mean more defects found, but with quickly diminishing returns [300], and two-person inspections can already be sufficient [56]. Many experiments assess review results by effectiveness (share of found defects) and efficiency (defects found per unit of time). But many of the early publications already stressed the value of inspections for individual and organizational learning, and Section 4 shows that secondary benefits are indeed considered in practice. These are just a few of the empirical results on reviews, and there are several published literature reviews and other overviews that provide a more detailed picture [17, 216, 302].

The given empirical results cast doubts on some of the assumptions of Fagan Inspections, but more serious problems are caused by the changes in development practices since their publication. Classic inspection is rooted in staged, document-centric development processes, but most current development is highly iterative or even 'continuous'. Several recent case studies indicate that a new style of 'modern' code review has replaced it [318]. These studies are confirmed with a larger sample in this thesis (Chapter 4). The modern style of reviewing code is adapted to continuous development by reviewing code *changes*, and it relies on computerized tools to gather the needed information. This, often informal, change-based style of reviewing was popularized by open-source practices. Peer review practices in open source software development have been studied intensively in the last decade, with contributions for example by Asundi and Jayant [15], Rigby and Storey [321], Wang et al. [395], Thongtanunam et al. [374] and Baysal et al. [44]. A survey by Bosu and Carver studied the impact of code review on peer impression among developers in open source projects [58]. Recent studies that assess code reviews in industry are more rare. Before this thesis, several case studies described its use for a limited number of cases. The semi-systematic literature review that is described in Section 3.2 identifies such studies. Baker [19] gave an early description of a change-based code review process in the industry and Bernhart et al. [50] describe its use (under the term "continuous differential code review") in airport operations software. Other small-scale studies of code review and inspection practices in the industry have been performed by Harjumaa, Tervonen, and Huttunen [162] and by Kollanus and Koskinen [204]. The latter study describes software inspection practices based on interviews with practitioners from five Finnish companies. In their sample, code review was quite rare and consequently not described in much detail. They conclude stating a need for further case studies on characteristics and problems of software inspection in practice. In Rigby's and Bird's study from 2013 [318], the authors compare peer review processes from several projects and note convergence towards a common process. This process is lightweight, flexible and based on the frequent review of small changes. Their analysis contains qualitative and quantitative parts, with a focus on the quantitative analysis of data sets from review tools. They surveyed a broad range of projects, but their study is limited to projects from large companies (Google, Microsoft, AMD) and large open source projects (Apache, Subversion, Linux, . . .). The studies in Part I of this thesis differ from theirs by using a different methodology and extend it by studying a broader range of organization sizes and styles. Nevertheless, Rigby's and Bird's study is closely related and this thesis confirms many of their findings.

The most recent academic survey on the state of review practices was published by Ciolkowski, Laitenberger, and Biffl in 2003 [75, 214]. This survey targeted not only code review, but also

reviews in other lifecycle phases. Its authors found a share of 28% of the 226 respondents using code reviews. A recent survey on software testing practices by Winter, Vosseberg, and Spillner [403] briefly touches upon reviews and notes that 52% of the respondents often or always perform reviews for source code. In a more specific survey, Bacchelli and Bird [18] studied expectations regarding code review at Microsoft and found a set of intended effects similar to the ones found in this thesis for a larger and different sample (Section 4.3).

Looking beyond the academic literature, there are some more recent surveys that contain information on code review practices. A whitepaper written in 2010 by Forrester Consulting [127] for Klocwork, a company selling a code review tool, notes that 25% of the 159 survey respondents use a review process that fits the definition of "regular, change-based code review" (Section 4.1). A survey performed in 2015 by Smartbear [354], another company selling code review software, contains information on code review practices and perceptions on code quality from about 600 respondents. Like the Forrester study, it contains very little information on the sampling method and possible biases. It states that 63% of their respondents perform tool-based code reviews.

$$\Diamond$$

All in all, contemporary code review practices have considerably departed from classic Fagan Inspection. Many results for Inspections might still hold, whereas others need to be adapted. The IEEE Standard on Reviews and Audits [179], for example, is not an adequate definition of common current code review practices. Therefore, Chapter 4 introduces definitions of code review that better suit today's reality.

3

Methodology to Assess the State of the Practice

As could be seen from the previous chapter, there is a lack of recent and rigorous research on the current state of code review use in industry. This part of the thesis aims to close this gap by means of several inter-related studies: An interview study with industrial software developers provides in-depth knowledge of their review processes, a systematic literature review augments these findings with data from other studies, and a large scale online survey tests and extends some of the claims from the first two studies. The current chapter describes the research methods used in these studies, so that the later chapters can build upon this knowledge when presenting the results. Many of the results from the studies were also published separately [37, 38, 39], and these publications contain further methodological details on the studies. Figure 3.1 shows the interplay of the research methods.

The research questions that are studied in the next chapters all revolve around the question of how code review is performed in industry today. For all of the RQs, the question 'What is the role of tools in this regard?' looms in the background. RQ_4[1] establishes the common ground:

> $\mathbf{RQ_4.}$ Which commonalities exist between code review processes of different teams and companies?

A positive effect of more efficient review tools could be to convince more development teams to use reviews. In this regards, RQ_5 studies the reasons for review adoption and cessation:

> $\mathbf{RQ_5.}$ Why are code reviews used or not used by industrial software development teams?

The studies find a lot of variation in the details of the review processes. To further lay a foundation for better review tools, it seems advisable to systematize these variations, and to determine what influences the teams' decisions:

---

[1]Throughout the thesis, the identifiers of RQs contain the chapter in which they are answered. So RQ_4 is answered in Chapter 4, and RQ_1 does not exist.
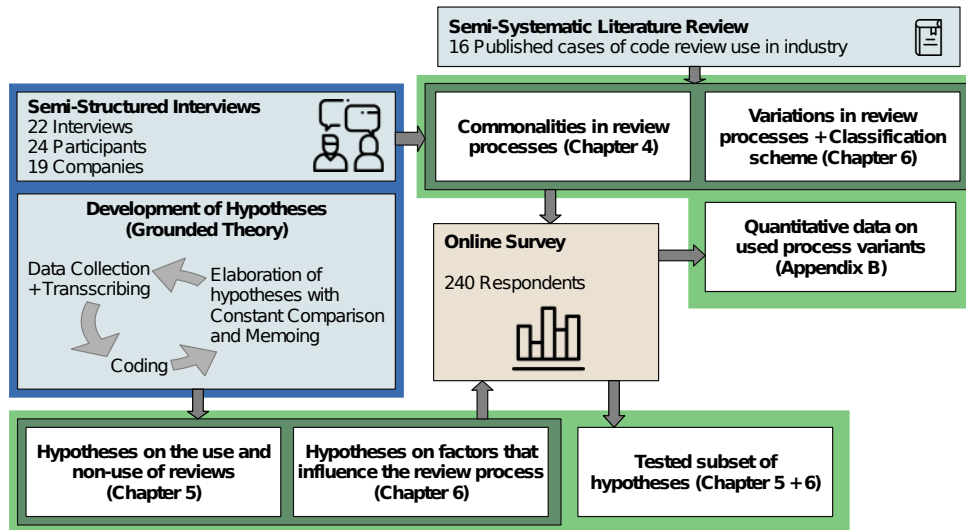
Figure 3.1: Overview of the data sources and results used to assess and explain the state of the practice

RQ$_{6.1}$**.** How can the variations between code review processes of different teams and companies be systematized?

RQ$_{6.2}$**.** Why are code reviews used the way they are?

3.1 Grounded Theory Interview Study

Besides determining the current state of the practice, this part of the thesis shows hypotheses regarding the factors influencing review process decisions and the role of tools. The *Grounded Theory* methodology [2, 81, 144] suits the goal of empirically generating such hypotheses well: It uses an iterative approach to build a theory that is 'grounded' in data. Interviews with industrial practitioners as the primary source of data allow to elicit motivations, opinions and detailed descriptions. To clarify the used interpretation of Grounded Theory, the chosen method is described in more detail in the following. Besides presenting the Grounded Theory methodology, this section also presents demographics on the interviewees and their companies. The articles on the Grounded Theory study were co-authored by Olga Boruszewski (Liskin), Kai Niklas, and Kurt Schneider. Raphael Pham gave further methodological advice.

I performed 'theoretical sampling' to select the interviewees: The emerging theory helped to choose participants that could extend or challenge that theory, for example because they came from a so far unexplored context. I gained access to the participants either by direct or indirect personal connections or by approaching them on conferences or after they showed interest in code reviews on the Internet. In total, the study is based on 22 interviews with 24 participants. They describe 22 different cases of code review use in 19 companies. Detailed information on participants' demographics and contextual information can be seen in Tables 3.1 and 3.2. The sample has a focus on small and medium standard software development companies and in-house IT departments from Germany, but it includes contrasting cases for all main factors. As some examples, company IS is much larger and company II much smaller than the other cases. Companies IB and II do the main development outside of Germany. The team from company IJ

Table 3.1: Demographics and review use of the companies from the interviews

ID[1]	Type[2]	Employees (IT, ca.)	Country	Development Process	Spatial	Regular Review Use
IA	in-house IT, travel	450	DE	agile	co-located	no
IB	std. softw., dev. tools	400	CZ	ad hoc	distributed	yes
IC	std. softw., government	200	DE	classic / ad hoc	co-located	no
ID	std. softw., CAD	100	DE	ad hoc	co-located	yes
IE	std. softw., output mgmt.	70	DE	agile	co-located	yes
IF	std. softw., agriculture	130	DE	agile	co-located	yes
IG	std. softw., retail	50	DE	agile	co-located	yes
IH	contractor, automotive	70	DE	agile	co-located	yes
II	SaaS, dev. tools	5	US	ad hoc	distributed	no
IJ	in-house IT, finance	1100	DE	ad hoc	co-located	no
IK	in-house IT, finance	200	DE	classic	co-located	no
IL	in-house IT, finance	400	DE	classic	co-located	yes
IM	in-house IT, government	200	DE	classic	co-located	no
IN	in-house IT, marketing	50	DE	agile	co-located	no
IO	–	–	DE	–	co-located	yes
IP	in-house IT, finance	–	DE	–	distributed	yes
IQ	in-house IT, retail	120	DE	classic / agile	co-located	yes
IR	in-house IT, marketing	50	DE	agile	co-located	no
IS	in-house IT, automotive	4000	DE	agile	co-located	yes

[1] All company IDs from the interviews start with 'I' to be able to separate them from IDs from other data sources that will be introduced later.

[2] For consultants, the company given is the consultancy's customer, not the consulting company itself.

does not directly work on a product (like most of the other cases), but on an architectural platform. The interviewees are mostly software developers and team or project leads, because the development teams were responsible for code reviews in the sampled cases. Company IE is the partner company that is described in more detail in Chapter 8. By sampling several interviewees from that company, differences in the descriptions between participants could be tested and more knowledge on the problems of individual reviewers gained. The second topic is addressed in the later parts of this thesis. The interviews were conducted between September 2014 and May 2015. Data collection was stopped when theoretical saturation was reached, i.e., when there was only marginal new information in the last interviews.

The interviews are semi-structured, using open-ended questions. The corresponding interview guide was initially created based on the research questions and checked by another researcher (Pham) who has experience with interview studies and Grounded Theory. It was later continually adjusted according to earlier interview experiences and the emerging theory. The interviews lasted 46 minutes on average, ranging from 24 minutes to 78 minutes. Face-to-face

Table 3.2: Demographics of interviewees

Company	ID	Role	Industrial Softw. Dev. Experience (Years)
IA	1	software developer	30
IB	2	software developer	15
IC	3	software developer	15
ID	4	software developer	17
IE	5	team/project lead	10
IE	6	software developer	25
IE	7	software developer	10
IE	8	software developer	7
IE	9	software developer	7
IE	10	software developer	6
IF	11	team/project lead	12
IG	12	team/project lead	10
IH	13	team/project lead	15
II	14	team/project lead	14
IJ	15	software developer	16
IK	16	software developer	3
IL	17	software developer	6
IM	18	requirements engineer (consultant)	20
IN	19	software developer (consultant)	20
IO	20	team/project lead (consultant)	–
IP	21	team/project lead (consultant)	–
IQ	22	software developer	3
IR	23	software developer	14
IS	24	team/project lead (consultant)	18

interviews were preferred; Skype or telephone were used for 5 interviewees where a face-to-face interview was not possible. Three participants (IDs 19, 20, and 21 in Table 3.2) were interrogated in a group interview[2], all other interviews were conducted with single persons. All interviews were recorded and later transcribed. To reduce the risk of bias when interviewing colleagues in company IE, the respective interviews were performed by another researcher (Boruszewski).

After performing and transcribing the first four interviews, I started data analysis: I used open coding to identify common themes in the data and analyzed the resulting codes for dimensions in which they vary as well as for similarities. Coding was done paper-based at first and later using the CAQDAS software Atlas.TI [131]. Coding was done incrementally and iteratively, including new interviews as they were taken and revisiting most interviews several times. In this constant comparison process [81], I compared and related citations and codes to each other. Furthermore, I contrasted whole cases of review usage to carve out their differences and similarities. The basic open coding was done again indepently by a second researcher (Niklas). The results were compared and discussed afterwards to check for possible bias or different view-

---

[2]The gate-keeper which had provided access to these interviewees left the company during the study. They could not be reached by other means. Therefore, some entries in Table 3.1 and 3.2 could not be determined and had to be left blank.

points. During the whole process, memos were written to capture emerging ideas. The resulting analysis was reported back to all participants, asking for review regarding misunderstandings and relevance. This "member checking" resulted in minor extensions and changes to the theory and increased the confidence that the results are a suitable description of the participants' reality.

3.2   Systematic Literature Review

In addition to the interviews, a semi-systematic literature review is performed to find recent descriptions of industrial code review processes. The found descriptions are used to triangulate and extend the findings from the interviews. The literature review is systematic in the way that it uses the rigorous procedure for snowballing-based systematic literature reviews described by Wohlin [405]. It is semi-systematic because the decisions regarding inclusion or exclusion of studies were done only by a single researcher.

The inclusion criteria for studies were as follows: (1) the study has been published since 2006 (inclusive), (2) it has been peer-reviewed, (3) it is published in English and (4) it describes code review practices in industry in some detail. There has to be some indication that the process is really used and not only brought into the company for a case study by the researchers. The description of the code review process does not have to be the article's main topic, as long as it is described in enough detail. Open source projects that are largely driven by a company (e.g., Android, Qt) are included as "industrial", other open source projects are excluded. The review in my original article [38] contained publications up to 2016. For this thesis it is extended up to the start of 2019.

The start set consists of four papers: [50, 266, 318, 342]. They have been chosen because they span a number of different years and publication venues. In addition, the article by Rigby

Table 3.3: Companies with review process information extracted from the literature review

ID	Company name	Sources
LE	Eiffel Software	[266]
LA	AMD	[312, 318]
LX	*name unknown*	[248]
LC	Critical Software S.A.	[122]
LF	Frequentis	[49, 50]
LV	VMWare	[20]
LM	Microsoft	[18, 55, 60, 61, 241, 318]
LG	Google/Android	[275, 318]
LQ	Digia/Qt	[257, 375]
LL	Salesforce.com	[413]
LS	Sony Mobile	[342]
LN	Vendasta	[310]
LY	*name unknown*	[104]
LH	Shopify	[207]
LO	Google	[331]
LD	Dell EMC	[396]

and Bird [318] is a key publication that combines several previous studies and that is cited quite often. Saturation was reached after four iterations. Table 3.3 lists the found sources, grouped by the companies whose code review process they describe. The information gained from the publications was much more shallow compared to the rich descriptions from the interviews. Its main use is as additional evidence for the identified variants.

3.3   Online Survey

Theoretically sampled interviews can provide in-depth data, but they require a lot of effort to get a broad overview and are not suitable to test hypotheses. Therefore, a survey was performed in a third step, based on the results of the interview study and the literature review. The survey was joint work with Hendrik Leßmann: He created the survey instrument and conducted the survey as part of his master thesis [232].

The survey's target population consists of all commercial software development teams. As there is no repository of all software development teams, a controlled random sampling of participants was not possible. Instead, they were contacted via a number of communication channels: We (Baum, Leßmann) directly contacted 32 people belonging to the target population from our personal networks. We further asked 23 people outside the target population from our networks to advertise the survey. We posted the survey invitation to several online communities, on Twitter, Xing, and Facebook; and also advertised the survey at a German software engineering conference. Finally, we posted the invitation on some mailing lists. Probably the most important single channel was a post on the mailing list of the German software craftsmanship communities ("Softwerkskammer"), reaching out to roughly 1400 people. When selecting channels, we took care to avoid introducing bias on the type of review process used. Specifically, we decided against sampling GitHub users, and we turned down an offer to spread the invitation to a mailing list of former participants of a series of review courses.

The intended granularity of sampling was teams. As the survey was conducted anonymously, it was not possible to tell whether two respondents came from the same or different teams. Survey participants were informed that we only want one response per team. When inviting participants directly, we took care to only invite one person per company. Nevertheless, there is a risk that the sample includes several respondents from the same team.

Most parts of the survey were created based on the results of the interviews and literature study. The process of survey creation followed established guidelines [183, 363]. To ease answering and analyzing the survey, it mainly contains multiple-choice and numerical questions. Following guidelines for survey research [363], it was tried to reuse questions from existing surveys, but only a limited number of questions from the first version of the HELENA survey [211] could be reused with adjustments.

The questionnaire was iteratively tested and refined. Initial internal testing was followed by six rounds of pre-tests, four of these with members of the target population and two with PhD students. The final survey also allowed the participants to enter feedback on the survey, which was checked for possible problems. There was a German as well as an English translation of the questionnaire.

It became evident early during questionnaire creation that if all hypotheses and factors from the interview study were included, the survey would become too long for the intended audience. Therefore, the scope of the questionnaire was limited and the questionnaire was split into a main part and an optional extension part. Answering the main part took around 15 minutes

and answering the extension part additional 8 minutes in the pre-tests.

The survey questions can be roughly classified into four groups: (1) Demographics or filter questions (e.g., on the country, role of the participant or the use of reviews), (2) questions on the context of the review process (e.g., product, development process, team characteristics, ...) (3) questions on the used review process (based on the classification scheme in Appendix B) and (4) ranking questions to assess the relative importance of intended and unintended review effects. The full instrument can be found in the online material of the original publication [36].

The survey data analysis constitutes a mix of descriptive and inferential statistics. Multiple-choice questions that contained an 'other' option with free-text answers were coded for analysis.

Most statistical tests performed during analysis checked for a dependence between two dichotomous variables. Unless otherwise noted, these 2x2 contingency tables were checked using Fisher's exact test and statistical significance was tested at the 5% level. Bonferroni correction is used when there are multiple tests for a research question, but not between research questions. Confidence intervals for proportions are 95% confidence intervals calculated using the Clopper-Pearson method. All percentages are presented rounded to the nearest integer. All but the filter questions were optional, to avoid forcing participants to answer. The resulting missing data was handled by 'pairwise deletion' (also called 'available case analysis'), i.e., participants were excluded only from those analyses where data was missing for at least one of the needed questions.

Data collection happened in early 2017. In total, 240 respondents from the target population answered the survey. 130 participants went on to answer the extension part after finishing the main part. The respondents are working in 19 different countries. The majority of respondents, 170 (76%), is from Germany. 33 respondents (15%) work in other European countries, 11 (5%) in Asia (including the Middle East) and 11 (5%) in Northern America. 19 respondents (10%) were invited directly by one of the researchers, 30 (16%) were indirectly invited by other people, 104 (55%) heard about the survey on a mailing list, 24 (13%) in an online forum and 13 (7%) named some other channel. When asked about their role, 154 respondents (67%) said they mainly work as a developer, 50 (22%) work as architects, 14 (6%) as managers and 11 (5%) gave other roles.

The survey's target population is teams in commercial software development, so the large majority (94%, 215 teams) of the responding teams works on closed source software. The remaining share (14 teams) said their team mainly works on an open source project. The teams
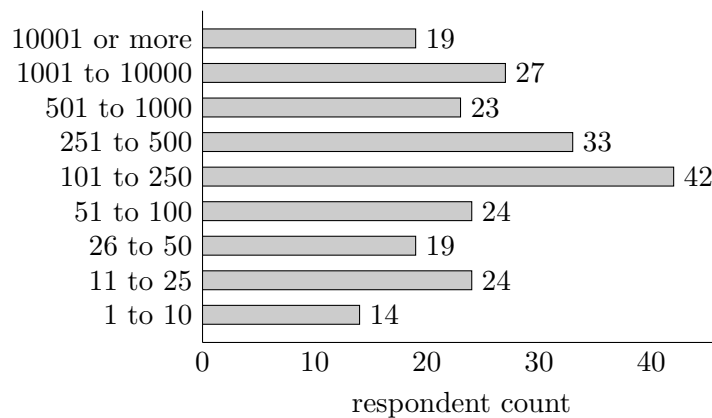

Figure 3.2: Company sizes (number of employees) in survey

19

work in companies of vastly differing sizes, from less than 10 to more than 10,000 employees; Figure 3.2 shows the detailed distribution of company sizes. 68% (148 of 217) of the participants work in co-located (as opposed to distributed) teams.

3.4   Validity and Limitations

This section presents the limitations of the claims put forward based on the interview study, the survey and the results from the literature. They are presented jointly because the three studies complement each other.

The interview study relies on the human researcher as instrument for data collection and analysis and is therefore prone to researcher bias. To mitigate this threat, the study follows Grounded Theory best practices, a reflexive research approach [71], and uses additional measures described below.

There is a risk that the interviewees left out or changed some aspects of their processes when describing them. This risk is reduced because the interviews did not touch upon sensitive personal data and interviewees were assured anonymity. The validity of the interview data could also be threatened by asking for descriptions and rationales after the fact. There is only one data point for most of the companies, and a larger study using several data points from each company as well as longitudinal observation could provide more reliable results.

During data analysis, there is the risk of introducing observational bias, for example when important data is not taken into account or open questions are not followed up. This risk is reduced through the used Grounded Theory practices: Careful and thorough coding, constant comparison, theoretical sampling and memoing. To avoid premature closing of the interviews, the interviewer explicitly asked for points missed in the discussion at the end of each interview. By recording and transcribing all interviews, it was ensured that no information was lost unintentionally and that the data could be coded and checked several times.

To mitigate researcher bias during data analysis and interpretation, coding was performed by two researchers (Baum, Niklas) and the results were discussed by all four authors of the publication. Another important mitigation for observational as well as researcher bias was 'member checking': I provided the results to the study participants and asked for feedback, which was then incorporated into the study.

As Grounded Theory studies use theoretical sampling and rather small sample sizes, it is hard to assess their generalizability. The study used a broad range of different interviewees from a heterogeneous sample of companies, and the interviewed sample is large compared to most qualitative studies. The most significant biases are that most of the companies and all of the interviewees are German, and that none of the interviewees came from a team producing highly safety critical software or from upper management.

Regarding the literature review, I mitigated many potential threats to validity by following the rigorous procedure outlined by Wohlin [405]. A snowballing-based literature review is sensitive to the choice of the start set. I chose a start set that was heterogeneous and quite large, which minimized this threat. The choice to only include peer-reviewed publications ensured a minimum of reliability of the data. It comes at the cost of excluding many review process descriptions from other sources, e.g., from blogs on the Internet. The most significant threat to the validity of the literature review is that the decision on inclusion and exclusion was done only by a single author.

The data sources combined cover various different review contexts. Nevertheless, the set of

potential code review processes is essentially unbounded, so it would be presumptuous to claim that the study captures *all* variation there is in change-based industrial code review processes.

For the survey, the primary threat to internal validity is sampling bias, given that the survey was distributed over various channels without the possibility to control who answered. Consequently, the participants likely differ systematically from the population of all developers, and they do so not only in their geographical distribution: They are probably more interested in code reviews and/or in process improvement or software quality in general. Due to this bias, the share of teams using code review in the survey should be regarded as an upper limit rather than an estimate of the real proportion. Apart from this bias, it was actively tried to avoid favoring certain types of code review processes in the sample.

A general problem of online surveys is that there is little control over the quality of responses. The survey included filter questions to check whether participants belong to the target population. Another threat with long online surveys is survey fatigue. As 209 of 240 participants reached the end of the main part, there is no indication of major fatigue effects.

The survey was anonymous, and most of the questions did not touch upon sensitive topics. However, the results of some questions might be influenced by social desirability bias, e.g., by stating that the team is using reviews just because it is desirable to do. Again, this might have influenced the descriptive results.

A weakness of the chosen method of data collection, i.e., of cross-sectional observational studies, is that they cannot be used to distinguish between correlation and causation.

$$\diamondsuit$$

Summing up, the state of the practice is assessed with a combination of three data sources: Semi-structured interviews in 19 companies, a semi-systematic literature review that provides data on 16 additional companies, and an online survey with responses from 240 development teams. Besides the descriptive results, the interview data is used to develop hypotheses on reasons for the current state, which are then partly tested with data from the survey.

4

Reviews in Current Industrial Practice

Before delving into the details of the processes in later chapters, this chapter delineates the common ground and addresses RQ_4: "Which commonalities exist between code review processes of different teams and companies?" The practice of *change-based code review* is defined based on the interview data. Results from the survey confirm that it is the dominating review variant in practice. Another important foundation are the effects that teams try to reach when using code reviews, and they are discussed at the end of the chapter.

4.1 Commonalities of Review Processes in Practice

This section presents the definitions of code review and change-based code review that were derived based on the interviews. All interviewees have a rather broad but common idea of code reviews. It is summarized in the following definition:

Definition 1 (Code Review)**.** *Code Review* is a software quality assurance activity with the following properties:
- The main checking is done by one or several humans.
- At least one of these humans is not the code's author.
- The checking is performed mainly by viewing and reading source code.
- It is performed after implementation or as an interruption of implementation.

The humans performing the checking, excluding the author, are called "reviewers".

Each part of the definition delimits code review from other quality assurance techniques, namely static analysis, self checks, testing and pair programming. All these delimitations are blurred: Human reviewers can be assisted by static code analysis, they sometimes execute tests or click through the GUI, in some cases the author joins the reviewers in checking his own code, and when author and reviewer jointly correct issues on-the-fly, they are basically doing pair programming. Definition 1 specifies a least common denominator of what practitioners consider a code review, in contrast to the definitions given in the IEEE Standard for Software Reviews and Audits [179] that describe specific processes in detail.

In the interviews, there were cases in which code review is performed irregularly and driven by individual needs, as well as cases in which there is a defined and regularly used code review

process. This thesis focuses on reviews that are done regularly. In all of the observed cases, this regular code review process is change-based:

Definition 2 (Regular, change-based code review). *Regular, change-based code review* is a type of code review that is codified in the development process of the team or organization in the following way: Every time a 'unit of work'[a] is seen as 'ready for review', all changes that were performed in the course of its implementation are considered a review candidate. This candidate is then assessed: For which parts of the change is a review needed, and is it needed at all? If a review is needed, the review candidate then waits for the reviewers to start reviewing.

---

[a]The term "unit of work" is similar to the 'patch set' identified in other studies. The definition does not use the term 'patch set' because that term is more narrowly focused on specific technologies.

The following commonalities of the observed change-based code review processes were identified from the interviews:

- No management action is required to trigger single code reviews, they are triggered solely based on pre-agreed rules. This replacement of the planning phase with conventions and rules is the difference most consistently separating regular change-based code reviews from classical inspection variants.
- When code reviews are performed in addition to unit testing or other developer-centric tests, testing is performed before code review. The same applies to static code analysis and to (other) checks that are performed automatically on a continuous integration server.
- The number of reviewers is two or less for the majority of reviews.
- All teams try to prevent situations in which code review happens after the changes are released to the customer.

4.2 The Dominance of Change-Based Code Review

The interviews suggest that change-based review is very frequent, and so does the study done by Rigby and Bird [318]. But results from a large scale survey were missing so far. This section provides quantitative empirical support for the dominance of change-based reviews in practice and also studies the prevalence of several more specific review styles. Finally, the further claims on convergent practices by Rigby and Bird are tested.

Of the companies from the interviews, eleven (*IB*, *ID - IH*, *IL*, *IO*, *IP*, *IQ*, and *IS*) have a regular, change-based code review process, whereas the remaining eight (*IA*, *IC*, *II*, *IJ*, *IK*, *IM*, *IN*, *IR*) only do irregular code reviews. Systematic review that is not change-based is mentioned in the interviews, but always in the form of "we did this once, but it was discontinued". In the literature, *LE* and *LC* describe cases that do not use change-based code review. All other literature sources describe change-based code review processes.

In the survey, the participants were asked how the review scope is determined: Based on changes, based on architectural properties of the software (whole module/package/class) or in some other way (with free text for further details). With a share of 90% (146 of 163; confidence interval 84 – 94%) of the teams doing code reviews, a change-based review scope is indeed dominating.

In the literature on code review and related work practices, there are slightly differing definitions and descriptions of sub-styles of change-based code review. Table 4.1 shows the frequency

24

Table 4.1: Frequency of use of different styles of code review

Style	Used Approximation of Definition Using Survey Constructs	Frequency of Use
Review based on code-changes	scope=changes	90% (146/163)
Regular, change-based code review (Definition 2)	scope=changes and trigger=rules	60% (96/160)
Contemporary Code Review [318]	scope=changes and publicness=pre-commit and unit-of-work≤user-story	46% (61/133)
Pull-based software development [147]	scope=changes and trigger=rules and publicness=pre-commit and interaction=no-meeting	22% (29/134)
Approximating Inspection [117]	interaction=meeting and communication=oral+stored and temporal-arrangement=parallel and trigger=explicit	2% (3/141)

of use for "modern/contemporary code review" [318], "regular, change-based review" (this thesis) and "pull-based software development" [147]. As not all of these sub-styles are concisely defined in the respective publications, the table also shows how the definitions/descriptions were approximated in terms of constructs used in the survey. Most of the teams that do not fall under Rigby and Bird's description of contemporary code review do so because they do not use pre-commit reviews (pre-commit: 46%, 61 teams; post-commit: 54%, 72 teams). There is only one respondent whose team uses a review scope that is larger than a user story/requirement.

The survey did not focus on Fagan-style Inspection [117] and, therefore, the current analysis cannot tell for sure whether a team uses a fully-fledged Inspection process to review code. To estimate an upper bound on the number of teams doing Inspection, a number of necessary conditions that would hold for those teams (see Table 4.1) were combined. Only 2% (3/141; confidence interval 0 - 6%) of the teams have a process that approximates Inspection in that way. Because much existing research on modern/change-based code review is based on open-source development or agile teams, I also checked whether there is a difference in the use of change-based review between open-source and closed-source products and between agile and classic development processes. There is no statistically significant difference in either case.

Apart from their description of a change-based review process as referred to in Table 4.1, Rigby and Bird consolidated three further convergent review practices:

1. "*Contemporary review usually involves two reviewers. However, the number of reviewers is not fixed and can vary to accommodate other factors, such as the complexity of a change.*" [318]: The survey results support the finding that the usual number of reviewers is low, indeed the numbers are even lower than Rigby and Bird's.[1] The average usual number of reviewers in the sample is 1.57, the median is 1 reviewer. With regard to the accommodation of other factors when determining the number of reviewers, 51% of the teams (47 of 92) named at least one rule that they use to adjust the number of reviewers in certain situations. The most commonly used rule is to decrease the number of review-

---

[1]The numbers are not fully comparable: Rigby and Bird looked at the actual number of reviewers in a large sample of reviews, whereas the survey asked for the usual number of reviewers in a review.
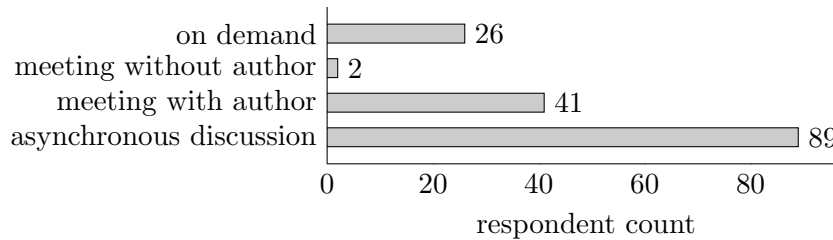
Figure 4.1: Interaction during Reviews

ers or to skip code review completely when the code change was implemented using pair programming: Such a rule is used in 36% of the teams.

2. *"Contemporary reviewers prefers [sic] discussion and fixing code over reporting defects."* [318]: Figure 4.1 shows how the surveyed teams usually interact during a review. Depending on how many of the teams discuss code during review meetings, between 55% and 81% of the teams have a review process that includes discussion of the code. Regarding fixing the code, 54% (84 of 157) of the respondents indicate that reviewers sometimes or often fix code during a review. This pragmatic attitude towards the classic boundaries of code review also shows up when 76% (69 of 91) of the respondents state that the reviewer executes the code for testing during review at least occasionally.

3. *"Tool supported review provides the benefits of traceability, when compared to email based review, and can record implicit measures, when compared to traditional inspection. The rise in adoption of review tools provides an indicator of success."* [318]: In the survey sample, 59% of the teams (96 of 163) use at least one specialized review tool. 33% (54 of 163) use only general software development tools, like ticket system and version control system, for review. 13 respondents indicated no tool use.[2] The ability of specialized review tools to record implicit measures might be one of their benefits, but it is seldom used in practice. Only 5% (4 of 88) of the teams systematically analyze review metrics. Further details on tool use are given in Chapter 7.

4.3 Desired and Undesired Review Effects

Reviews are used not only for finding defects, but for a variety of reasons. This section presents these desired effects, and also the undesired potential side-effects. The described effects were mentioned in the interviews as generally relevant to a team or organization as a whole. Besides these, there are also effects mostly relevant to single developers (e.g., "solving a specific problem"). This thesis does not describe those single developer effects, as they are of minor importance for regular code reviews. An overview of the team level effects is shown in Table 4.2.

Better code quality (desired) Reviews are expected to have a positive effect on internal code quality and maintainability (readability, uniformity, understandability, etc.). This effect results most directly from checking and fixing the found issues. Furthermore, the interviewees claim a preventive effect which leads developers to take more care in writing code that will be reviewed. Lastly, a better distribution of knowledge increases uniformity (see "Learning" below).

---

[2]A weakness in the used questionnaire is that there was no explicit "We do not use any tool" choice available. Therefore, the distinction between non-response and non-use of tools cannot be reliably made.

Table 4.2: Overview of found desired and undesired review effects, with results from the survey.

Type	Effect	Rank in Survey		
		Mean	Mode	Distribution
Desired	Better code quality	2.24	1 (37%)	■■▬▬——|
	Finding defects	2.79	1 (35%)	■■▬▬▬▬▬
	Finding better solutions	4.09	5 (21%)	▬■■■■▬▬
	Learning (author)	4.38	5 (23%)	▬▬■■■▬▬
	Learning (reviewer)	4.49	6 (28%)	▬▬■■■■▬
	Sense of mutual responsibility	4.72	6 (21%)	▬▬▬▬■■■
	Complying to QA guidelines	5.29	7 (45%)	▬▬▬▬▬▬■
Undesired	Increased cycle time	1.79	2 (41%)	■■▬
	Staff effort	2.08	2 (36%)	■■■
	Offending the author / Social problems	2.13	3 (45%)	■▬■

Finding defects (desired) Reviews are expected to find defects (regarding external quality aspects, especially correctness). This is seen as particularly important for defects that are hard to find using tests, such as concurrency or integration problems.

Learning of the reviewer (desired) Reviews are expected to trigger learning of the reviewers: They gain knowledge about the specific change and the affected module, but also more general knowledge on the coding style of the author and possibly new ways to solve problems. In this way, regular review shall lead to a balancing of skills and values in the team.

Learning of the author (desired) Reviews are expected to trigger learning of the authors: They get to know their own weaknesses (sometimes as simple as unknown coding guidelines). Furthermore, they learn new possibilities to solve certain problems, for example using libraries they did not know about: *"You just don't develop [better skills] if other people don't look at [your code]."*[I.12][3] Additionally, they learn something about the reviewers' values and quality norms for source code. Like the previous point, regular review shall consequently lead to a balancing of skills and values in the team.

Sense of mutual responsibility (desired) Reviews are expected to increase a sense of collective code ownership and to increase a sense of solidarity: *"...that it's not a single person's code, but that we strengthen the feeling of having a common code base"*[I.10]

Finding better solutions (desired) Reviews are expected to generate ideas for new and better solutions and ideas that transcend the specific code at hand.

Complying to QA guidelines (desired) There can be external guidelines that demand code reviews or even certain styles of code reviews. Such guidelines may be safety regulations and standards or customer demands in the case of contractors.

Staff effort (undesired) Performing reviews demands an investment of effort that could be used for other tasks.

Increased cycle time (undesired) Performing reviews increases the time until a feature is regarded "done". This increase can be split into the time needed until the review starts, the time for the review itself and finally the time until the found issues are corrected.

---

[3]Interviewee IDs are from Table 3.2

Offending the author (undesired) The author can feel offended (or discouraged) when his or her code is reviewed or when issues are found. Although this effect is mostly undesired, one of the interviewees explicitly mentioned a case where an individual used reviews as a form of bullying and intended to offend the author. A related effect occurs when the reviewers refrain from noting certain issues because they fear offending or discouraging the author: *"Then unfortunately you always have to give them so many review comments. Then one always feels bad, because you think they think 'they always beat me up'."*$_{I.10}$

As stated at the start of this section, the above-mentioned effects are based on the interviews. In a study at Microsoft, Bacchelli and Bird [18] also studied reasons for performing code reviews and found a similar set of intended effects.

$$\diamondsuit$$

Summing up, this chapter provides the definitions for *code review* and its subtype *regular, change-based code review*. It shows that review based on code changes is dominating in practice. Developing better tool-based support for these processes is the main focus of this thesis. A second contribution that is reused on several occasions is a set of common goals that teams usually try to reach with reviews.

5

Use and Non-Use of Reviews – Culture Beats More Efficient Tools

Review tools can help teams that already use code reviews. But another option to improve the state of the practice would be to have teams that currently don't use reviews introduce them for their benefit. This section presents the results on RQ$_5$: "Why are code reviews used or not used by industrial software development teams?" It turns out that making reviews more efficient will probably have little impact on review adoption, as the main influence is the company's and team's culture.

5.1   Triggers of Review Introduction

This section deals with the causes for the introduction of code reviews. According to the interviews, introduction of reviews was done mostly as a reaction to a problematic incident: There had been quality issues regarding correctness or maintainability $_{I.2,5,7,10,12,21,24}$ (*"It started this way: There was a mail from the project manager that there are these [quality] problems, and that we want to solve them [with reviews]."* $_{I.2}$), quality standards in the team were diverging $_{I.11,13}$ or some changes were perceived as risky or insecure $_{I.4,21}$. Another trigger is demand for code reviews by an external stakeholder $_{I.19}$, or positive experiences with reviews in the past $_{I.13,24}$. In many cases, a single developer initiated a discussion on the introduction of reviews in the (at least partly self-organizing) team, which was followed by an 'experimental' introduction of reviews $_{I.4,7,24}$: *"Some day we realized: The code doesn't look the way we want it to. I don't remember it exactly, I think we did it in the retrospective, but it could have been in a discussion at the coffee maker, too. Where we said: We have to do something, and then somehow reviews came to our mind."* $_{I.24}$

On the other hand, when no problem was perceived, there was reluctance to introduce reviews $_{I.4,13,15,18}$: *"It's just this way, ... everybody has his tasks, and when it works it's fine. And you don't take the time to do [reviews]."* $_{I.15}$; *"Eventually, because everything works quite well, ... there's no need for action."* $_{I.4}$.

"Perceived problem" means that there was a gap between some goal and reality. The goal is influenced by a role model (like another team/project $_{I.2,13}$ or a professional movement like

'clean code development' $_{I.10}$), the product/project context $_{I.2,4,8,18,20}$ (e.g., 'Which quality level is needed for the customer?'), the team members' personalities $_{I.1,11,13,18}$ and the team's culture $_{I.1,2,5,11,15,16,18}$.

Generally spoken, the introduction of code reviews or changes to the code review process are done when (and only when) this topic area moves into the focus of the team or its management as being ripe for improvement. In addition to an ad hoc reaction to a perceived problem, this can also be triggered by a continuous improvement process $_{I.7,11,24}$ (e.g., agile retrospectives).

> **Hypothesis 5.1.** Code review processes are mainly introduced or changed when a problem, i.e., a gap between some goal and reality, is perceived.

5.2 Inhibitors of Review Introduction

In cases where regular reviews were not used, the interviewees named several factors that inhibited their introduction. The most basic case is when there is nobody who could review the code, either because there is no other person in the project or because the culture fosters a strong separation of responsibilities $_{I.9,10,15}$.

Another category of inhibitors are problems that are generally associated with change: There is resistance to change among the people concerned $_{I.16}$. Lack of knowledge and corresponding insecurity $_{I.16,18}$ belong to this category, too. Furthermore, performing a process change consumes effort, leading to conflicts with other projects and tasks $_{I.12,18}$. The effort for a process change varies widely and can be substantial in bureaucratic organizations $_{I.18}$.

More specific to reviews, there is a weighing of the desired and undesired effects. One important factor is the fear of social problems: A general fear of being rated, or fear of annoying certain (key) developers $_{I.1,18}$: *"I don't know it definitely, but what I hear again and again, and the impression I get is: The people have fear of others looking at their code and telling them they did it badly."* $_{I.1}$; *"With developers that are in the business for a long time, it's difficult. You often have the attitude that it's their code, it belongs to them, and you shouldn't meddle with it."* $_{I.18}$ Where fear of social problems is not dominating, there remains the needed time and effort facing the expected benefits $_{I.4,15,22,23}$.

> **Hypothesis 5.2.** When code reviews are not used at all, this is mainly due to cultural and social issues. Needed time and effort are another important, but secondary, factor.

Hypothesis 5.2 can be triangulated with data from the online survey. I determined the subset of factors that are the best predictors for the distinction between teams that use reviews and teams that have not used reviews so far. The selection was done using Weka's 'CfsSubsetEval' [404]. Based on the resulting set of seven factors, a logistic regression model was built with R [308]. Table 5.1 shows the respective model statistics. Four of the influential factors can be considered as aspects of the team's or company's culture: A defined (not 'ad hoc') development process, use of static analysis, a preferred knowledge distribution of generalists instead of specialists, and a positive error culture. It may seem questionable to consider "use of static analysis" as a cultural factor, but the decision is backed up by a principal component analysis of the contextual factors. The two main dimensions of the results can be seen in Figure 5.1. In this analysis, "use of static analysis" is similar to aspects of quality and long-term orientation and orthogonal for example to the severity of defect consequences. Nevertheless, the reader should

Table 5.1: Regression coefficients and analysis of deviance statistics for a logistic regression model to predict review use vs non-use. Factors written in *italic* are cultural factors. All factors are binary.

Factor	Coefficient	Deviance	Resid. Df	Resid. Deviance	p (χ^2)
(Intercept)	-4.3604		106	105.968	
Development process not "ad hoc"	1.7857	11.5957	105	94.372	0.0007
Use of static analysis (code quality culture)	2.0575	14.7884	104	79.584	0.0001
Team size 5 or larger	1.6489	6.5211	103	73.063	0.0107
Preference to "generalists" instead of "specialists"	1.2725	4.8392	102	68.223	0.0278
Type of software is neither "games" nor "research"	18.4912	5.1486	101	63.075	0.0233
Positive error culture	2.4576	5.2481	100	57.827	0.0220
Team works spatially distributed	0.7652	0.7472	99	57.079	0.3874
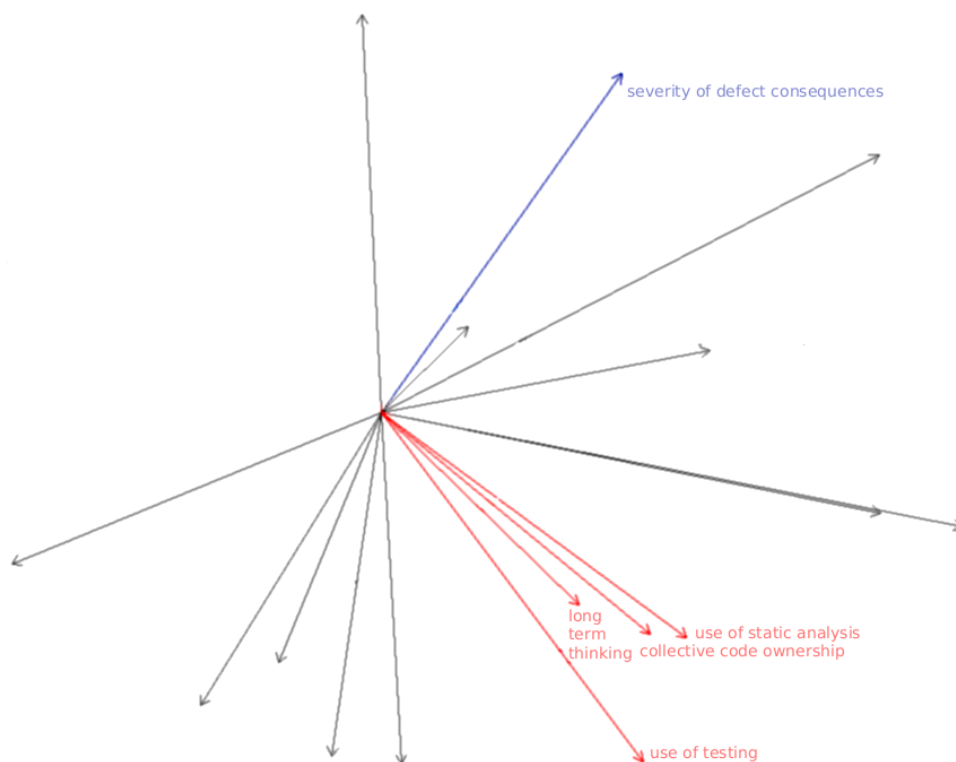
Figure 5.1: Principal Component Analysis of the contextual factors based on the survey data (projection on the two main components). Use of static analysis is similar to various aspects of quality orientation and orthogonal to defect consequences in the shown main dimensions. Unlabeled arrows are other factors from the survey.

keep in mind that principal component analysis and stepwise factor selection are not able to show a direct causal relationship.

Performing reviews normally decreases fear of social problems as well as the problems themselves: Fear is reduced due to habituation, the ability to accept criticism is increased and the corporate feeling is improved $_{I.11,24}$ (*"[There have been problems], right at the start, when [reviews] were introduced. Especially here, where we develop software for fifty years and have some old veterans. ... But this learning process has advanced considerably, and now the positive effects of a code review prevail for everybody."*$_{I.11}$). The needed time and effort does not decrease as much. This could explain the observation that although culture and social problems are the top reason to refrain from starting reviews, a negative assessment of costs versus benefits dominates when reviews are stopped again $_{I.4,19,20}$.

Hypothesis 5.3. The importance of negative social effects decreases with time when reviews are in regular use.

When reviews are stopped, this has most often been described in the interviews as *"fading away"*$_{I.1,3,7,13,19}$ and was seldom an explicitly stated directive. The survey asked the teams whether they are currently using code reviews. Teams not using code reviews were subdivided further: Have they never used reviews before, and if so have they never thought about it or did they explicitly decide against review use? Or did they stop using reviews in the past, and if so was this an explicit decision or did the review use "fade away"? Figure 5.2 shows the results: With a share of 78% (186 teams), the majority of teams is currently using code reviews.[1] 38 teams (16%) have never used code reviews so far, 8 of them because there was an explicit decision against their use. In 16 teams (7%), the use of code reviews ended, but in only one of those teams this was an explicit decision.

The risk of fading away seems to increase when developers have to perform a conscious decision every time they want to get a review. When taking this decision, there are immediately observable costs, while many benefits materialize only in the long term. This fits to the observation that the effects with personal short-term benefits dominate in the cases doing irregular reviews $_{I.3,4,14,15,23}$. When reviews are institutionalized, the risk of fading away is reduced: A fixed integration of reviews in the process and a tool that supports this process impedes decisions against reviews, and regularly performing reviews leads to routine and habituation. This also fits the observed cases, where all teams doing frequent reviews have institutionalized them in their development process.

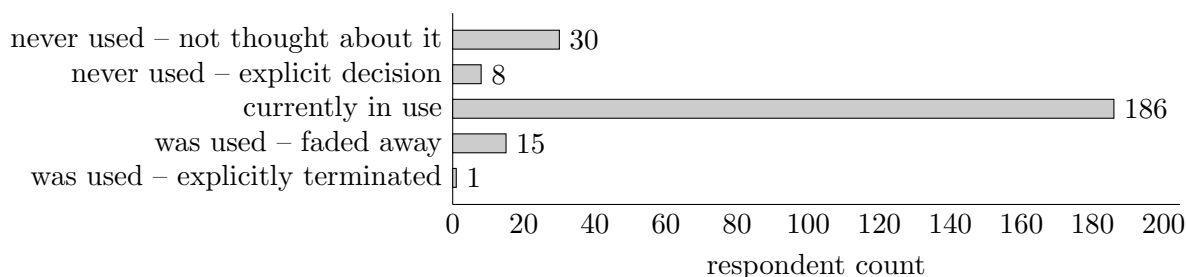[1]This number is likely biased, see Section 3.4.

Figure 5.2: Survey results on use of reviews and reasons for non-use

Table 5.2: Review triggers vs review continuation

Trigger	Reviews in use	Review use faded away
Manager	7	3 (30%)
Reviewer	14	2 (13%)
Author	38	4 (10%)
Rules/conventions	103	3 (3%)

> **Hypothesis 5.4.** Code review is most likely to remain in use if it is embedded into the development process (and its supporting tools) so that it does not require a conscious decision to do a review.

The results from the survey can be used to test a subset of this hypothesis: *The risk that code review use fades away depends on the mechanism that is used to determine that a review shall take place: This risk is lower when rules or conventions are used instead of ad hoc decisions.*

To test this hypothesis, two sub-samples from the survey are compared: Teams currently doing code reviews, and teams where review use faded away. The survey also asked how it was decided whether a review should take place: By fixed rules or conventions, or ad hoc on a case-by-case basis. For the ad hoc triggers, the survey further distinguished triggering by the reviewer, the author or a manager. There was the possibility for respondents to select "other" and enter a free-text description.

Of twelve teams that answered this question for which review use faded away, three used rules or conventions and the remaining nine used ad hoc decisions. For the 162 teams currently using reviews, the relation was 103 with rules/conventions compared to 59 without. Put differently, the risk to be in the "fade away" subsample increases from 2.8% with rule triggers to 13.2% with ad hoc triggers, a risk ratio of 4.7. The exact Fisher test of the corresponding 2x2 contingency table results in a p-value of 0.0124, therefore the difference is statistically significant at the 5% level. Table 5.2 shows the detailed numbers for the different review triggers. An interesting side-note is that having managers trigger reviews seems to be especially prone to discontinuation.

Another possible explanation for the higher share of teams with ad hoc triggers in the "fade away" subsample would be a generation effect: Teams that introduced reviews more recently could use rule triggers more often. Therefore, teams that have used reviews for less than a year were compared with those that used them for two years or more. Of 45 teams with brief review use, 25 use rules (56%). For teams with long review use, the share is 49 of 75 (65%). This higher share of rule use for longer review use supports Hypothesis 5.4 and opposes the stated generation effect.

5.3 Comparison to Related Work

The found results confirm some previous studies and challenge or extend others. The current section discusses the results in comparison to related work as well as to more general theories.

Harjumaa et al. [162] studied characteristics of and motivators and demotivators for peer reviews. The study is based on twelve development divisions from ten small Finnish software companies. Interviews were performed in a structured fashion, based on a questionnaire developed using information from the literature, and analyzed mostly quantitatively. The authors examined motivators for review, with defect detection being the most important one. The only

obstacle they identified as relevant is lack of time and resources. They did not examine cultural or social issues in depth.

In his book "Diffusion of Innovations" [324], Rogers describes a general theory of the mechanisms behind the spread of new ideas and improvements, based on a large body of empirical work from several disciplines. Diffusion is described as a process of communication in social networks. Others' opinions and experiences are most important when deciding if to adopt, but they also influence the knowledge of innovations. The rate of adoption of an innovation is influenced by several of its characteristics: Relative advantage compared to current solutions, compatibility to values, experiences and needs, complexity, trialability and observability.

Code reviews are part of the general software development process, and introducing code reviews is a special case of software development process improvement. Therefore, it is interesting to compare the results to general studies on this topic. In Coleman's study of software development process formation and evolution in practice [79, 80], he observed that process change is triggered by "business events", often problems. This is similar to Hypothesis 5.1. After a change, "process erosion" occurs until a "minimum process" is reached. In the studied cases, major erosion occurred only when code reviews had not been institutionalized (Hypothesis 5.4). Like this thesis, Coleman found that software process is influenced by cultural issues ("management style", "employee buy-in", "bureaucracy", ...) as well as business issues ("market requirements", "cost of process"). However, in his area of research cost of process was dominating, while this thesis found a dominance of cultural and social issues among inhibitors of review use. This can possibly be explained by the more social and subjective character of code reviews.

Some other sources also support Hypothesis 5.4: Komssi et al. [205] describe experiences at F-Secure where document inspections were not integrated into the process and were abandoned as soon as the champions lost their interest. Land and Higgs [219] describe the institutionalization of development practices in a case study of an Australian software company.

$$\diamondsuit$$

Summing up, this chapter studied to what degree better review support could lead to higher review adoption, based on empirically grounded hypotheses on why reviews are introduced and stopped. Non-introduction of reviews is mainly influenced by cultural and social issues and fit to the context, so that better review support is unlikely to increase review introduction much. Stopping of reviews, instead, is often not an explicit decision but a 'fading away'. By using rules and conventions to trigger reviews, the risk of 'fading away' can be substantially reduced. This is a strong point of tool-supported change-based review processes.

6

Variations in Industrial Review Processes

Although industrial review processes are converging, there is a lot of variation in the details. The next section answers $RQ_{6.1}$ ("How can the variations between code review processes of different teams and companies be systematized?") and presents a model to classify these variants. A single dimension is not adequate to capture the variation. Instead a multi-dimensional model is used, based on several so called 'facets' of the process. But where does the variation come from, and which factors determine the review process? The later parts of this chapter address $RQ_{6.2}$ by discussing these questions, and compare the results obtained from the empirical data with findings from the literature.

6.1   A Faceted Classification Scheme of Change-Based Code Review Processes

Common classification schemes for review processes use simple labels like "Inspection", "walkthrough" or "passaround" to classify types of review processes as a whole. Such a one-dimensional taxonomy simplifies discussions, but it is inadequate for describing the variations without losing a lot of information. Therefore, this thesis proposes a *faceted* classification scheme: A review process is described by the combination of values for a number of facets/dimensions. Because the data led to a large number of facets, the scheme is grouped into thematic categories. Figure 6.1 shows the groups, the contained facets and the possible values for the facets. The thematic categories in the classification scheme are:

Process embedding The first group of facets contains aspects that vary with regard to how the code review process is embedded into the rest of the development process, i.e., when and in which way a review is triggered and which influences it has on other process activities.

Reviewers The facets belonging to this group describe differences regarding the selection of reviewers.

Checking The facets in this group describe variations regarding the central activity of a code review: Checking the code. Like in the rest of the scheme, only variations in the codified processes of the teams are described. Personal differences in the checking habits of the individual reviewers are out of the scope of the classification scheme.
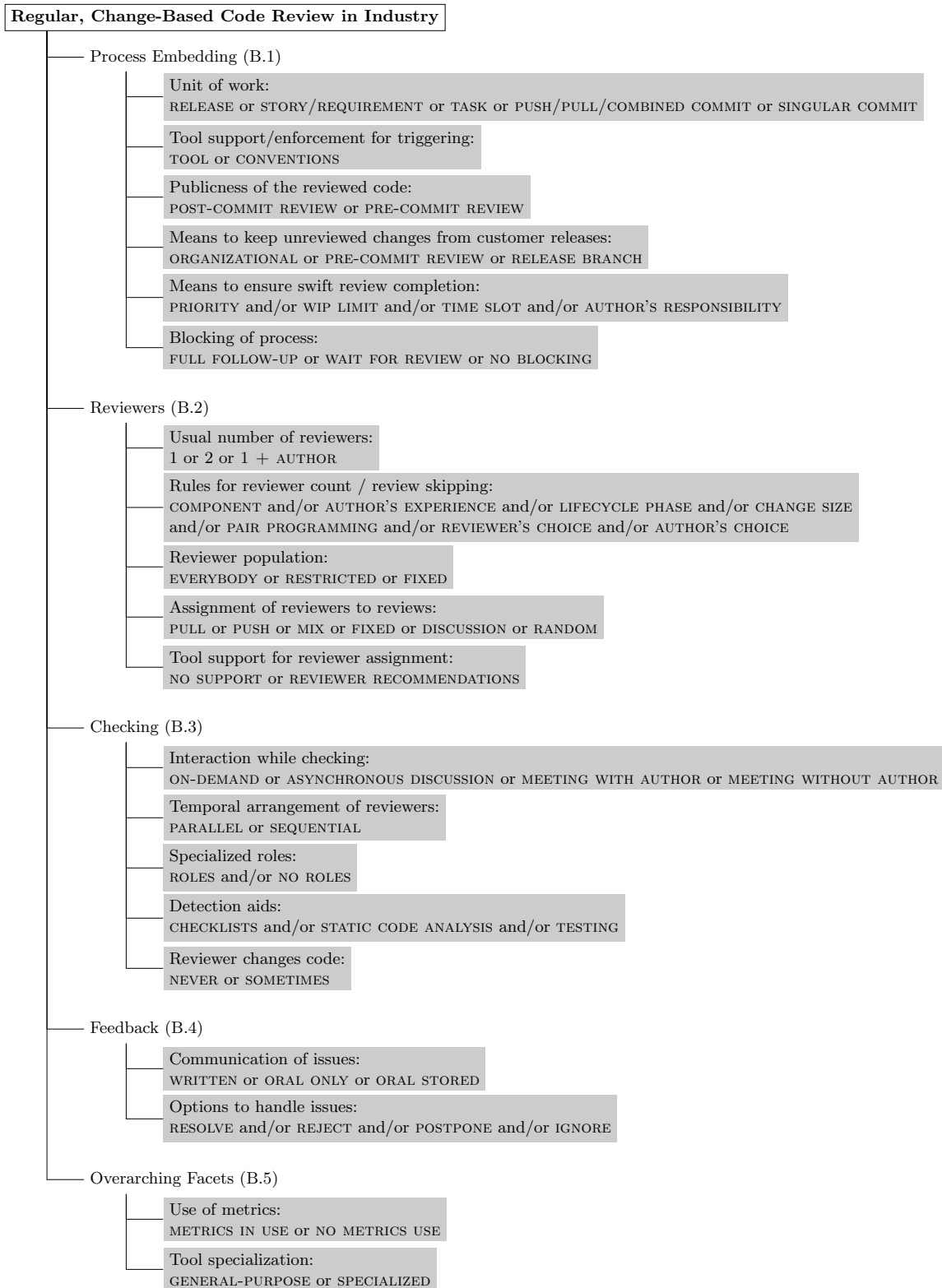
Regular, Change-Based Code Review in Industry

— Process Embedding (B.1)

Unit of work:
RELEASE or STORY/REQUIREMENT or TASK or PUSH/PULL/COMBINED COMMIT or SINGULAR COMMIT

Tool support/enforcement for triggering:
TOOL or CONVENTIONS

Publicness of the reviewed code:
POST-COMMIT REVIEW or PRE-COMMIT REVIEW

Means to keep unreviewed changes from customer releases:
ORGANIZATIONAL or PRE-COMMIT REVIEW or RELEASE BRANCH

Means to ensure swift review completion:
PRIORITY and/or WIP LIMIT and/or TIME SLOT and/or AUTHOR'S RESPONSIBILITY

Blocking of process:
FULL FOLLOW-UP or WAIT FOR REVIEW or NO BLOCKING

— Reviewers (B.2)

Usual number of reviewers:
1 or 2 or 1 + AUTHOR

Rules for reviewer count / review skipping:
COMPONENT and/or AUTHOR'S EXPERIENCE and/or LIFECYCLE PHASE and/or CHANGE SIZE and/or PAIR PROGRAMMING and/or REVIEWER'S CHOICE and/or AUTHOR'S CHOICE

Reviewer population:
EVERYBODY or RESTRICTED or FIXED

Assignment of reviewers to reviews:
PULL or PUSH or MIX or FIXED or DISCUSSION or RANDOM

Tool support for reviewer assignment:
NO SUPPORT or REVIEWER RECOMMENDATIONS

— Checking (B.3)

Interaction while checking:
ON-DEMAND or ASYNCHRONOUS DISCUSSION or MEETING WITH AUTHOR or MEETING WITHOUT AUTHOR

Temporal arrangement of reviewers:
PARALLEL or SEQUENTIAL

Specialized roles:
ROLES and/or NO ROLES

Detection aids:
CHECKLISTS and/or STATIC CODE ANALYSIS and/or TESTING

Reviewer changes code:
NEVER or SOMETIMES

— Feedback (B.4)

Communication of issues:
WRITTEN or ORAL ONLY or ORAL STORED

Options to handle issues:
RESOLVE and/or REJECT and/or POSTPONE and/or IGNORE

— Overarching Facets (B.5)

Use of metrics:
METRICS IN USE or NO METRICS USE

Tool specialization:
GENERAL-PURPOSE or SPECIALIZED

Figure 6.1: Overview of the classification scheme. Each gray box is a facet. Possible values are written in small caps. Values separated by "or" are alternatives, values separated by "and/or" can be combined. A tuple with values for all facets describes a review process. (based on [38])

Feedback Feedback in the form of review remarks is the main output of the checking. The facets in this group summarize differences in the handling of feedback between the team's review processes.

Overarching facets The facets in this group pertain to aspects that span the whole code review process, for example the use of review tools.

The details of the various facets are not described in the current chapter, but are available in Appendix B.

6.2 Factors Shaping the Review Process

Observing the variation in review processes, the question of the sources of this variation comes up. This section presents hypotheses on factors that shape the review process. But it also describes weaknesses of these hypotheses, as they could not be fully validated in the online survey. An important partial result for the rest of the thesis is that model processes are an important factor shaping the details of the process, and that review tools are an important source for these model processes. Scientific publications, on the other hand, seem to have little direct influence on practice.

When a change in (or introduction of) the review process is triggered, a small number of possible solutions is examined $_{I.5,7,17,24}$ (see also Section 6.2.3). Each possibility has to satisfy mainly three criteria: It has to fit into the context of the team $_{I.4,5,9,12,17,24}$, and it has to be believed to provide the expected desired effects $_{I.3,6,8,9,10,14,19,21,22,23}$ while staying in the acceptable range of undesired effects $_{I.4,6,7,9,10,14,18,19,21,22}$. These general relationships are visualized in Figure 6.2. The term "effect level" shall indicate that the team is looking for a solution that is "good enough" with regard to the desired effects and "not too bad" with regard to undesired effects $_{I.10,12,13}$. Influences between the contextual factors are not included in the diagram.

The factors forming the context are divided into five categories: "culture", "development team", "product", "development process" and "tool context". The following text describes the factors extracted from the interviews. It also gives examples for their influence on the review process.

The category "culture" subsumes conventions, values and beliefs of the team or company. The following inter-related factors belong to this category:

Collective Code Ownership In companies with full collective code ownership, every developer can and should work on every part of the code. At the other end of the spectrum are companies where only a designated module owner is allowed to change certain parts of the code. As an example, this directly influences whether a reviewer is allowed to fix minor issues on his own during the review $_{I.2,5,6,8}$. It also influences the importance of clarity of the code to other developers, which indirectly influences the intended level of code quality to be reached by reviews.

Intended knowledge distribution The intended knowledge distribution is closely related to collective code ownership. When every developer should be able to work on every part of the code, knowledge has to be distributed broadly. This increases the need for review as a means of knowledge distribution $_{I.11}$. Many interviewees believe that face-to-face communication is better suited for knowledge distribution than written feedback $_{I.4,7,8,9,11,21}$.
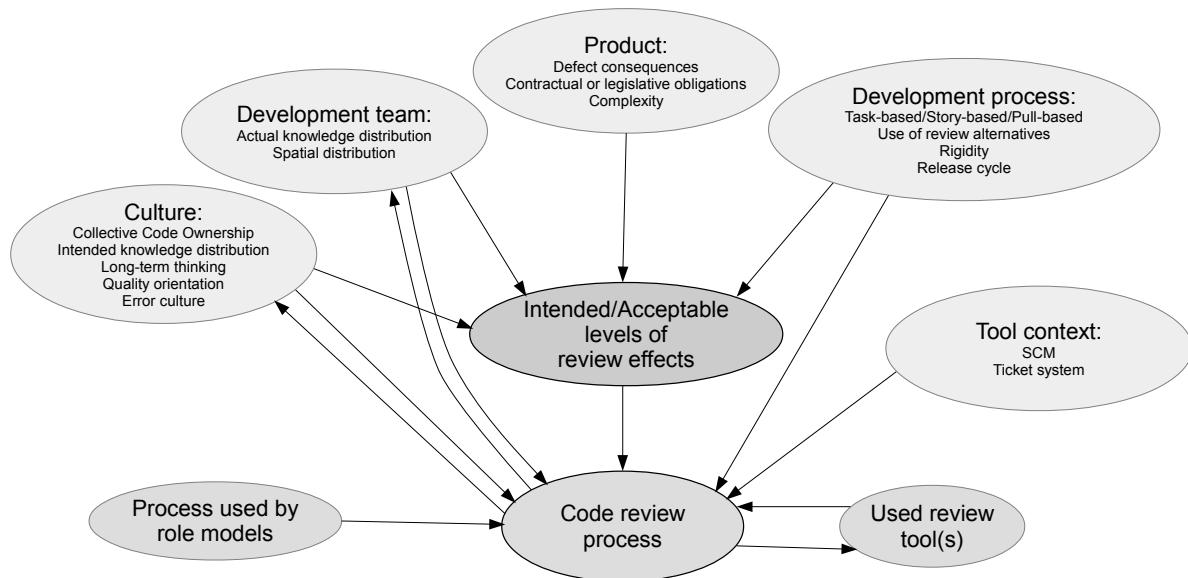
Figure 6.2: Main factors shaping the review process. Arrows mean "influences". (Source: [39])

Long-term thinking An orientation towards the long-term success of the company or product increases the importance of code quality and knowledge distribution and therefore influences review process choices indirectly [I.7,15].

Quality orientation The balance between quality, effort and time-to-market differs between teams and companies. When quality is considered to be of secondary importance, the effort and time spent on reviews becomes an important factor, and vice-versa [I.5,6,13,18,23,24].

Error culture When errors are seen as personal failures of the author, this increases the risk that the author feels offended by review remarks [I.12,17].

The category "development team" subsumes factors that characterize the development team:

Actual knowledge distribution The reviewer's expertise is seen as an important factor for its effectiveness in finding defects [I.2,4,5,9,11,14,22]. Therefore, some teams choose to restrict the reviewer population to experienced team members [I.12,22,24]. Less experienced team members often introduce more defects and benefit more from knowledge transfer through reviews [I.14,19,21,24].

Spatial distribution Some teams work co-located, others are distributed. In distributed teams, face-to-face interaction in reviews is harder [I.4].

The category "product" contains factors that characterize the requirements posed to the developed product. These factors have an indirect influence on the review process:

Defect consequences When defects have severe consequences, finding defects becomes more important. The importance of finding defects influences for example the number of reviewers and the selection of certain experienced reviewers [I.24].

Contractual or legislative obligations When code reviews are mandated by a contract or law, the review process is designed to satisfy these requirements [I.20].

Complexity When the developed code is not very complex, the intended levels of code quality and defects can possibly be reached without doing regular code reviews [I.8,18].

The code review process is a subprocess of the general development process and is influenced by the other parts of this process:

Task/Story/Pull-based Some teams divide user stories into development tasks, while others only use stories or only pull requests based on commits. A team can only choose between tasks and stories as the unit of work to review when both are available in the development process.

Use of review alternatives There are alternative techniques to reach the effects intended by code reviews. A commonly used alternative for defect detection is testing [I.6,12,14,20,23,24]. Many teams use static code analysis on a continuous integration server to detect maintainability issues [I.2,3,6,8,11,23,24]. To find better solutions, many teams discuss requirements and design alternatives before code review [I.5,6,7,23]. Pair programming is another alternative to code reviews, often seen to provide similar benefits [I.1,3,6,7,8,24].

Rigidity When the development process leaves a lot of freedom to the single developer, techniques like pull requests help to ensure that reviews are performed [I.2].

Release cycle When releases are very frequent, the acceptable increase in cycle time through reviews is lower [I.23]. Frequent or even "continuous" releases also demand techniques to keep unreviewed changes from being delivered to the customer. This often means precommit reviews/pull requests [I.2].

Tools used in the development team can reduce the possible choices of process variants:

SCM To use a pull-based review process, a decentralized source code management system (SCM) is needed [I.9,12,20].

Ticket system The review process can only be codified into the ticket system's workflows if a ticket system is in use and supports customizable ticket workflows [I.12].

6.2.1 Effect Goals as a Mediator

In the preceding section, it could already be seen that some contextual factors influence the review process directly by delimiting which process variants are feasible. Many others influence it indirectly by influencing the intended/acceptable levels of review effects ("..., *but which processes you introduce is heavily linked to how valuable you perceive them*"[I.14]; Figure 6.2). Based on the interviews, many process variants are expected to promote certain effects, and often also to impair others. This leads to conflicts between the effects. Consequently, the chosen review process is influenced by the combination of intended effects. Some effects are seen as more important than others, while others are seen as secondary or not pursued at all. This is used to perform trade-offs while designing the review process. The resulting combination is not constant for a team, but can be different for example for different modules or for different phases of the release cycle.

> **Hypothesis 6.1.** The intended and acceptable levels of review effects are a mediator in determining the code review process.

An effect can be desired by the team or by a single developer. Only effects that are desired by the team lead to a codification of the review process in form of a process specification or conventions. This then leads to a more homogeneous process, while the review practices stay inhomogeneous when they are driven largely by individual needs.

The process for selection of the review variants is not done comprehensively in most cases. Most interviewees argued with positive and negative effects of certain techniques but did not

include every effect into their consideration. And in some cases it seems that the primacy of a certain effect is taken for granted, without stating it explicitly $_{I.13,22,24}$.

Hypothesis 6.1 was also tested based on the survey results. Following this hypothesis, one would expect to find that the relative ranking of review effects influences the chosen variant for some of the review process facets, that the team's context influences the relative ranking of review effects, and that this indirect effect is in most cases stronger than the direct influence of context on review process facets.

I systematically checked each of the combinations of review effect and process facet that were mentioned in the interviews [27]. For intended review effects, none of the checked interactions were statistically significant, even at the 10% level and without Bonferroni correction. For the relative ranking of undesired effects, some of the predicted effects had p-values smaller than 0.05:

- When "increased staff effort" is most unintended this makes a "very small review scope" (i.e., more overhead due to a higher number of small reviews) less likely: risk ratio=2.2; p=0.034.
- When "increased staff effort" is most unintended this makes "pull or mixed reviewer to review assignment" more likely: risk ratio=1.6; p=0.037.
- When "increased cycle time" is most unintended this makes "review meetings" less likely: risk ratio=2.8; p=0.006.

Those three interactions are also those with the highest risk ratio (i.e., effect size). Even though they have p-values smaller than 0.05, none of them is statistically significant after Bonferroni correction. A complete list of all tested interactions can be found in the survey study's online material [36].

Summing up, only 3 of 30 cases give some support for the expected relationship. Therefore, there is little evidence that the intended and acceptable levels of review effects influence the code review process, except in some narrow areas. Consequently, they cannot be mediators, and Hypothesis 6.1 is not supported by the survey.

Due to the low statistical power and multiple threats, the analysis of this hypothesis is problematic. Assuming that the non-finding is not caused by flaws in the data collection and analysis, there are two explanations: (1) There is an effect, but the study checked the wrong sub-hypotheses, or (2) the intended effects determine a team's review process only to a small degree. The second explanation is in line with Ciolkowski, Laitenberger, and Biffl's conclusion that many companies use reviews unsystematically [75]. It would also mean that 'satisficing' [249] and orientation along experiences from peers and processes used by review tools are even more important than noted earlier. There remains a lot of research to be done, both to find out which process variants are best in a given situation, and to find ways to bring these results into practical use.

6.2.2 Sources of Information

Until here, it was mainly described how the choice among the possibilities is performed, but it remained open how the possibilities to examine are determined: The interviewees were asked which sources, if any, they used to gain knowledge on reviews. Most did not explicitly look for information on reviews very often and also perceived no need for further information $_{I.3,8,12,16,17,18}$. Own experiences in open source projects and experiences from colleagues were the most influential $_{I.2}$. Overall, the following information sources were mentioned:

- colleagues or other teams in the same company $_{I.2,13}$

- open source projects $_{I.9}$
- blogs and web pages $_{I.1,10,21}$
- university education $_{I.14}$
- practitioners' journals $_{I.1,15,18,24}$
- practitioners' conferences $_{I.24}$
- books on software engineering best practices $_{I.15,24}$

6.2.3 The Influence of Model Processes

In addition to the sources of information mentioned in the preceding section, many of the considered process variants were conceived without having an explicit source of information $_{I.12}$. When choosing a review process, the number of examined possibilities was often quite low $_{I.7,13,17,21}$. In many cases, only the possibility that first came to mind was examined, and only when it was not suitable to reach the intended effect levels a search for further possibilities was started $_{I.7,8,13,21}$.

This low number of examined possibilities is especially true for many of the minor decisions involved in the choice of a review process. Mostly, the potential for improvement of these process variations is regarded as minimal. Consequently, many of these minor decisions have been justified by the interviewees with statements like 'We tried it that way, it worked well enough, so there is no reason to change it' $_{I.10,12,13}$. This observation is similar to the one that led to Hypothesis 5.1 ("change only on perceived problem").

From another point of view, the observations described in the preceding paragraph also mean that the initial choice of a possibility for consideration has a tremendous effect on the final process. Among other things, this explains a pattern of 'tool shapes process' that occurred several times $_{IB,IE,IG,IH,IS}$: A team selects a certain tool to support its review process. This tool has certain 'standard' process variants. Consequently, the team mainly uses the standard variants and only tries to circumvent the limitations of the tool (or looks for another tool) when it expects a notable increase in effect level attainment or the standard is in conflict to fixed contextual factors.

> **Hypothesis 6.2.** Model processes known from other teams or projects or coming from review tools have a large influence on many minor decisions shaping the code review process.

6.3 Comparison to Related Work

This section discusses the results in comparison to the related work as well as to more general theories.

As already mentioned in Chapter 5, Rogers [324] describes a general theory of the mechanisms behind the spread of new ideas and improvements. Others' opinions and experiences are most important when deciding if to adopt, but they also influence the knowledge of innovations. This supports that review processes of familiar teams have a large influence (Hypothesis 6.2). The rate of adoption of an innovation is influenced by several of the innovation's characteristics: Relative advantage compared to current solutions, compatibility to values, experiences and needs, complexity, trialability and observability. Given these categories, the "intended review effects" coincide with the perceived "relative benefit", and the cultural issues inhibiting review use can be seen as cultural incompatibility. Rogers also notes that "re-invention", the customization of an innovation to one's needs, is another factor benefiting innovation adoption.

All the variations in the interview study are essentially re-inventions of the basic notion of code review.

Theories of 'bounded rationality', particularly 'satisficing', are used to explain decisions in organizations [249]. They claim that humans typically do not choose an optimal solution, but instead stop searching for further solutions when one is found that satisfies their needs. The findings support this theory: A team does not search for the optimal review process, but instead uses one that reaches the intended effect levels (Hypothesis 6.1).

Code reviews are part of the general software development process, and introducing code reviews is a special case of software process improvement. Therefore, it is interesting to compare the study's results to general studies on this topic. Clarke and O'Connor combined several studies to develop a reference framework of factors affecting the software development process [76]. Compared to the classifications from the current thesis, they provide a lot more structure regarding environmental factors ("Organization", "Application", "Business"), but cultural factors are mainly restricted to only two sub-categories ("Culture" and "Disharmony"). Further work in similar areas has been performed by Sánchez-Gordón and O'Connor for very small software companies [333], by Mustonen-Ollila and Lyytinen using diffusion of innovation theory as a guiding framework to quantify some aspects of the adoption of information system innovations [277] and by Orlikowski on the adoption of computer-aided software engineering tools [286].

Hypothesis 6.1 stated that the review process is tailored according to the pursued goals. Tailoring of review processes has also been proposed in the research literature, for example in the TAQtIC approach [100] with a focus on classical Inspection. And Green et al. [149] noted that "*[the pursued] value can change everything*" with regard to tailoring software processes.

Porter et al. [301] observed that only a small share of the variance in review performance can be explained by process structure, and Sauer et al. [336] arrive at a similar result by theoretical considerations. This indicates that the choice to give little attention to the review processes' details (Hypothesis 6.2) is likely a sensible one.

$$\Diamond$$

This chapter shows the large amount of variation in industrial review processes and proposes a classification scheme to systematize it. Furthermore, it provides grounded hypotheses on the factors that influence the used process. In Part II, Chapter 8 uses these findings when describing the partner company's context and review process. Another important hypothesis for this thesis is that there is a cyclic influence between review tool and used process – a chance and a responsibility for tool creators.

7

Tools and Techniques to Support Reviews

Researchers and practitioners alike have been looking for ways to support code reviews for decades. One possibility is to support the reviewer during the checking of the code. Much research in this regard was done under the label of 'reading techniques'. These reading techniques are introduced in the next section, as they are a predecessor of the computerized cognitive support techniques of Part III. Besides the checking, support is also possible for the bookkeeping and process management tasks. This is what most current code review tools focus on, and the last two sections of this chapter provide an overview of research on code review tools (Section 7.2) and of review tools used in current industrial practice (Section 7.3).

7.1 Code Reading Techniques

'Reading techniques' are a largely manual tool to support the reviewer. This section presents related work on reading techniques, and also discusses their limited use in practice. There is a multitude of reading techniques, and Table 7.1 shows an overview. Many of these reading techniques share underlying mechanisms that are assumed to be the cause for their functioning. These are:

Thoroughness Demanding that all artifacts of a certain type, or all artifacts satisfying certain conditions, are reviewed.

Focus/Prioritization The reviewer shall focus on parts that are more important, for example due to high business value or error proneness. Here, *focus* means a binary decision, whereas techniques that use *prioritization* distinguish several levels of priority, up to a total ordering of items.

Activation These techniques try to make the reviewer work actively with the document under review, for example by writing test cases for requirements.

Reduce Overlap By focusing different reviewers on different aspects, the overlap between multiple reviewers in a review team shall be reduced.

Include Documents The reviewer shall perform a comparison to other documents, for example to the requirements for a design review.

Step-by-Step The technique gives a detailed step-by-step procedure how to perform the checking, to make the process more repeatable and to guide inexperienced reviewers.

Table 7.1: Reading techniques and their main mechanisms.

Technique	Type	Proposed for	Thoroughness	Focus/Prioritization	Activation	Reduce Overlap	Include Documents	Step-by-Step	Secondary Outcome	Defect Cause Analysis
Checklist-Based Reading [65, 117, 165]	checklist	any		F						
Value-Based Review [231]	checklist	requirements		P						
Indicator-Based Inspections [114]	checklist	req./design	•	F			•	•		•
Rigorous Inspection Method [234]	checklist	design	•	F			•	•		
Abstraction-Based Reading [109]		code	•	F	•			•	•	
Functionality-Based Reading [1]	scenario	code	•	F	•		•	•		
Perspective-Based Reading [24]	scenario	requirements	•	F	•	•		•		
Traceability-Based Reading [378]	scenario	design	•	F	•	•	•	•		
Task-Directed Software Inspection [193, 194]		code	•		•			(•)	•	
Use-Case Based Reading [108]		code		F			•	•		
Usage-Based Reading [373]		design		P	•		•	•		
Time-Controlled Reading [298]		design		P	•		•	•		
Inspection-Based Testing [401]		design/code		F	•		•	•	•	
Test-Case Driven Inspection [113]		requirements		F	•		•	•	•	
Error-Abstraction and Inspection [12]		requirements		F						•

Secondary Outcome The outcomes of the reviewer's active work are meant to be useful on their own. This is usually used to lessen the burden of *Activation*. In 'task-directed software inspection', for example, the reviewers create documentation for the code under review.

Defect Cause Analysis The reviewer shall explicitly consider defect root causes, like common human errors.

There are two larger families of reading techniques, indicated by the 'Type' column in Table 7.1: Checklists are used to focus reviewers on certain aspects and to ensure that these are checked. Variants of scenario-based reading instead provide detailed descriptions how the reviewer should actively work with the document under review, often focusing different reviewers on different roles to reduce overlap. Not all details of the techniques can be discussed here, but a book by Zhu [414] and some earlier surveys [23, 25] provide in-depth discussions. Only few of the reading techniques in the table explicitly focus on code reviews, and none deal with code changes. Most reading techniques were developed in a time when review tools were in their infancy, and there is still little computer-support for most of them.

The survey on review use in practice also contained questions to roughly assess the use of the main families of reading techniques in practice. It is sometimes claimed that "checklist-based

reading" is the prevalent reading technique in practice. The survey results do not support this claim: Only 23% (22 of 94) of the respondents state that they use a checklist during reviews. Only 7% (6 of 90) of the respondents state that they explicitly assign distinct roles to the different reviewers. 72% (63 of 88) use neither checklists nor roles. Given the high research effort that went into developing reading techniques, this result is disappointing. The results from Chapters 4 to 6 hint at possible reasons:

- Reading techniques might not fit the context well: For example, assigning different roles to different people is only useful when there is more than one reviewer, but often there is only one. Creating tests during code review is useless when they are created during implementation. Requirements or design documents can only be reviewed when they exist, but many teams do not use them consistently. And none of the techniques is tailored for the review of code changes.
- The choice of review process is influenced more by experiences and tools than by research publications, and checklist-based reading is the only technique that is supported in at least some of the practically used tools (see Sections 7.2 and 7.3).
- Many experienced developers dislike the tedious manual work and strict procedures associated with many reading techniques. [226, 259]

All in all, this casts doubts on three of the mentioned mechanisms (*Reduce Overlap*, *Secondary Outcome*, and *Step-by-Step*) for regular change-based code reviews. Empirical results whether and under which conditions *Activation* is beneficial are inconclusive [99, 226, 402]. Of the remaining mechanisms, Chapter 9 shows how the review tool developed in this thesis (CoRT) supports *Thoroughness* and *Include Documents*. The identification of irrelevant change parts (Chapter 15) helps to *Focus* the reviewer. Support for *Defect Cause Analysis* is not discussed in detail in this thesis.

7.2 Research on Code Review Tools

This section presents the results of a semi-systematic literature review on code review tools. Such a literature review provides an overview of central innovations and the development of the field over time. It does not cover review tools not originating in academia, which form the majority of tools used in practice today and which are discussed in the next section.

In the literature review, an extensive start set of research papers was systematically completed by forward and backward snowballing [235, 405]. The start set ([48, 49, 53, 59, 62, 63, 64, 70, 74, 93, 94, 95, 106, 112, 141, 155, 156, 163, 167, 168, 174, 185, 189, 217, 224, 225, 237, 239, 296, 297, 335, 359, 370]) consisted not only of articles on review tools, but also of earlier literature reviews and overviews. Saturation was reached after three iterations. The review includes peer-reviewed publications that describe implemented code review tools, published until November 2018. In particular, this excludes (1) tools that are only described in Master or other student's theses (e.g., [274, 407]), (2) tools that explicitly target the review of non-code artifacts (e.g., [53, 151, 154, 156, 181, 209]), (3) descriptions how to apply general-purpose collaboration or code exploration tools to reviews (e.g., [11, 98, 266, 306, 385]), and (4) publications that only discuss concepts without at least a prototypical realisation (e.g., [103, 283]).

The literature review found 51 publications that describe 30 tools. They are shown in Tables 7.2 and 7.3. The earliest contribution, *ICICLE*, was also a substantial one and introduced many of the central ideas of review tools: Presentation of the code under review, automatic remark collection, and further support for process administration and bookkeeping. In addition, it

Table 7.2: Review tools originating from academia (1/2). The tools are ordered chronologically, except for successors of an earlier tool, which are marked by ↪.

Tool Name	Publ. Year	Artifact Type	Process Integration	Interaction	Evaluation	Major Innovations
ICICLE/ TRICICLE [62, 63, 64, 338]	1990	code	no	in-person (ICICLE), synchronous (TRICICLE)	industry	first published tool, code presentation, remark collection, process administration support, code cross-referencing, navigation to background knowledge, inclusion of static analysis review agents
InspeQ [201, 202]	1991	code	no	none	laboratory	support for 'phased inspections', checklist integration
CSI [106, 251]	1991	generic	no	synchronous	laboratory	review of non-code artifacts, support for geographical distribution
↪ CAIS [252]	1994	generic	no	asynchronous	laboratory	
↪ AISA [359]	1997	generic	no	asynchronous	industry	inspection of graphical artifacts
Scrutiny [141, 142]	1993	code	no	asynchronous or synchronous	industry	asynchronous
CSRS [185, 186]	1994	code	no	asynchronous	laboratory	collection of telemetry data for research
HyperCode [296, 297]	1997	code changes	no	none	industry	shows code changes, browser-based
ASSIST [237, 240, 270, 271]	1997	generic	no	asynchronous or synchronous	laboratory	configurable process, evaluation with controlled experiment, defect content prediction with capture-recapture
WIPS [217]	1998	code	no	synchronous	laboratory	
WiP [161]	1998	code	no	asynchronous	laboratory	
↪ WiT [160, 370]	1999	generic	no	asynchronous	laboratory	
↪ SATI [159]	2001	generic	no	none	laboratory	
↪ XATI [167, 168]	2002	generic	no	none	laboratory	
InspectA [269]	1999	code	no	asynchronous	laboratory	
AnnoSpec [358]	1999	code	no	asynchronous	laboratory	showing annotations from related code parts

Table 7.3: Review tools originating from academia (2/2). Names in *italics* are author names of otherwise unnamed tools.

Tool Name	Publ. Year	Artifact Type	Process Integration	Interaction	Evaluation	Major Innovations
IBIS [67, 222, 223, 224, 225]	2001	code	no	asynchronous	laboratory	
jInspect [372]	2004	code	no	none	laboratory	support for usage-based reading
ISPIS [188]	2004	generic	no	asynchronous	laboratory	reviewer recommendation
CIT [70]	2005	code	no	asynchronous	laboratory	support for scenario-based reading
Nick [282]	2005	code	no	none	laboratory	
WAIT [93, 94, 95, 96]	2007	generic	yes	asynchronous or synchronous	laboratory	process integration
CollabReview [304, 305]	2008	code	(no)	asynchronous	laboratory	continuous feedback
SCRUB [174]	2010	code	no	asynchronous and synchronous	industry	
ReviewClipse [48, 49]	2010	code changes	yes	asynchronous	industry	
CodeTalk [360]	2010	code	(no)	asynchronous	laboratory	
Fistbump [189]	2016	code changes	yes	asynchronous	laboratory	
Dürschmid [112]	2017	code	(no)	asynchronous	laboratory	
The Empire of Gemstones [335]	2017	code changes	yes	asynchronous	laboratory	gamification

contained features that are above the standard of many more recent tools, like support for code cross-referencing, presentation of background knowledge, and the use of review agents based on static code analysis. It was focused on Fagan-type Inspections, with mandatory synchronous review meetings that had to be held in the same room. *CSI*, published shortly afterward, was the first prototype tool to overcome this limitation and supported distributed review meetings. With the introduction of asynchronous discussion by *Scrutiny* and of explicit support for code changes in *Hypercode*, the features that define code review tools until today were established by the year 1997. Asynchronous discussion has become the standard since then, in line with the empirical findings from the 1990s that showed no substantial benefit of review meetings [187, 390].

Innovations also happened besides the main features: Certain publications brought support for reading techniques, like checklists in *InspeQ* and usage-based reading in *jInspect*, or other

extended support features, like support for re-inspection decisions in *ASSIST*. Further contributions are targeted towards research, like the collection of detailed telemetry data pioneered by *CSRS* or the evaluation with controlled experiments, first performed for *ASSIST*. Many later tools brought improvements in the details, adjustments to other technological bases, and occasionally also re-inventions of earlier concepts. Recently, there have also been several prototypes that try to overcome explicit reviews and move towards continuous feedback on code.

Whereas most early tools were stand-alone and not well integrated into the surrounding process, integration with SCMs, ticket systems and IDEs has become more common in the last ten years. Still, few tools from academia support reviews based on code changes, let alone fully integrated regular change-based code reviews as defined in Section 4.1. As Tables 7.2 and 7.3 show, tools that were developed or evaluated with industry are in the minority, but this minority was responsible for many of the central innovations.

7.3 The Use of Review Tools in Practice

The previous section presented a large number of review tools from academia. Now, it is shown which tools are actually used in practice. The survey on the state of the review practice (Section 3.3) asked development teams that perform reviews which review tools they use, if any. Figure 7.1 shows the most prevalent answers. 33% of the teams (54 of 163) only use general-purpose development tools during reviews. Among the specialized tools, the commercial or service offerings from Atlassian, GitHub, GitLab and JetBrains take the lion's share, with Gerrit as only non-commercial open-source contender. For the most prevalent tools (Bitbucket, GitHub, GitLab) the review support is only a sub-function of a general repository management and development support suite.

To compare these tools, Table 7.4 shows some of their central features. It contains all tools that were mentioned in the interviews or the survey, so that there is evidence that these tools are used in industry, or at least were used in the last years. The information on the tools was collected by trying them out, asking users, or looking up information from their websites. In contrast to the majority of tools with an academic origin, most of the tools analyzed here provide the integration with the SCM and the development process necessary to support regular change-based code reviews. Most tools are limited with regard to the process variants they support:
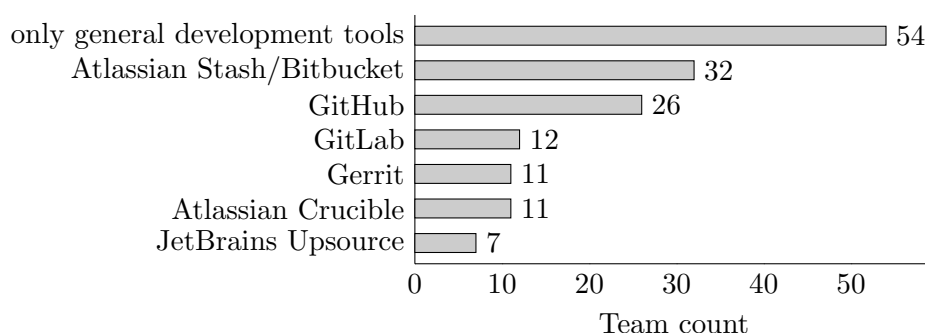
Figure 7.1: Number of teams that use a certain review tool. Multiple mentions were possible. The figure shows the most prevalent tools in the survey. Further mentions were: Team Foundation Server (3), SmartBear Collaborator (2), Phabricator (2), Codeflow (1), Reviewboard (1), ReviewAssistant (1), proprietary tools (4)

Table 7.4: Selected features of review tools used in practice.

Name[1]	Availability[2]	Unit of Work	Change-Based Integration				Reviewer Support	
			SCM Integration	Process Integration	Pre/Post-Commit	IDE Integration	Cross-Referencing	On-the-Fly Editing
Atlassian Stash / Bitbucket Server	commercial	pull request	git, Mercurial	yes	pre	no	no	yes
GitHub	SaaS	pull request	git	yes	pre	no[3]	no	yes
GitLab	SaaS	pull request	git	yes	pre	no	no	yes
Gerrit	open-source	pull request	git	yes	pre	possible (EGerrit)	with EGerrit	with EGerrit
Atlassian Crucible	commercial	review task	git, svn, …	no[4]	post	no	no	no
JetBrains Upsource	commercial	review task	git, svn, …	yes	post	optional	yes	no
Microsoft Team Foundation Server	closed-source	pull request	tfvc, git	yes	pre	yes	no	no
SmartBear Collaborator	closed-source	review task	git, svn, …	yes	both	yes	no	no
Phabricator	open-source	pull request	git, svn, …	yes	both	no	no	no
Microsoft Codeflow[5]	in-house	review task	tfvc, git, …	yes	pre	no	no	no
Reviewboard	open-source	review task	git, svn, …	no	both	no	no	no
ReviewAssistant	closed-source	pull request	git, svn, …	yes	pre	yes	no	no
ReviewClipse	open-source	review task	svn	no	post	yes	no	no

[1] Tools were chosen according to the interview and survey answers and are ordered by popularity.
[2] 'commercial' implies 'source available for paying customers' here; 'closed-source' is commercial without available source.
[3] as of IntelliJ IDEA 2018.3, there is a rudimentary support for GitHub pull requests in IntelliJ IDEA (https://blog.jetbrains.com/idea/2018/10/intellij-idea-2018-3-eap-github-pull-requests-and-more/)
[4] reviews cannot be made a mandatory part of the ticket/development workflow
[5] data for Codeflow might be outdated, as the tool is not publicly available

Many focus on pre-commit reviews, and few have explicit support for multiple review rounds in sequential review. Most of the tools are browser-based, and there is neither an IDE-integration nor an alternative support for advanced code navigation and editing. For many of the tools, the source code is not available or only available to paying customers, so the only way to adapt these tools is by using their plugin APIs, or by using hacks like changing the displayed HTML with a browser plugin. None of the tools provides the cognitive support features that are discussed in Part III, i.e., automatic change part ordering, advanced identification of irrelevant change parts and summarization of code changes.

$$\diamondsuit$$

Summing up, the discussion in this chapter shows the ideas underlying various code reading techniques, the core features that a review tool should possess, and also the limitations of current tools with regard to supported process variants, tool contexts and extendability for research. Section 7.2 also supported the design science research approach of developing innovative tools in the interplay between research and practice. Part II of this thesis now describes how the requirements of a specific company and the knowledge gained in Part I can be combined to build a state-of-the-art review tool and platform for code review research.

Part II

The Code Review Tool and Research Platform 'CoRT'

8

Context for Action Research on Improved Code Review Tooling: The Partner Company

The goal of this thesis is to improve code review tooling in industrial practice. To this end, it is of high value to be able to try out tool features in practice and to improve them iteratively. The collaboration with a software company provided the chance to do so. A description of this case-study context is needed for the discussion in later chapters, so the current chapter gives details on this company setting. In the chapters that follow, the tool and the decisions that led to its design are described.

The company is a medium-sized software company from Germany. It develops and sells a software product suite, as well as related consulting and services. Many of its customers are from the financial or public sector. The most severe consequence of defects can be financial losses for the customers. The company employs around 70 people, of which about 18 are full-time developers (growing from 14 in the year 2013, where the earliest empirical data used in the thesis is from). The thesis author also worked part-time for the company during his PhD studies and was a full-time employee before and afterwards.

In the year 2013, the SCM repository contained about 28,000 files and directories. Of these, about 7400 are Java files with a total of about 950,000 lines of Java code[1]. In 2018, the repository contained about 86,000 files and directories, of which about 20,000 are Java files with about 2.3 million lines of Java code. Java is the primary implementation language; only recently TypeScript was introduced as a second language for UI code.

The company's development team has been using an agile scrum/scrum-ban process since 2008. Currently, sprints last three weeks, and there is a public release of the product at the end of each sprint. There is a culture of collective code ownership, and the developers shall be able to work as generalists on many parts of the code base. The team works mostly co-located. Code quality is important, and code style, unit tests, and many other quality checks are performed on a continuous integration (CI) server. The development is done trunk-based in a monolithic Subversion repository. The company has a policy of a consistent development environment, for

---

[1]simply counting lines, not distinguishing between empty lines, comments, etc
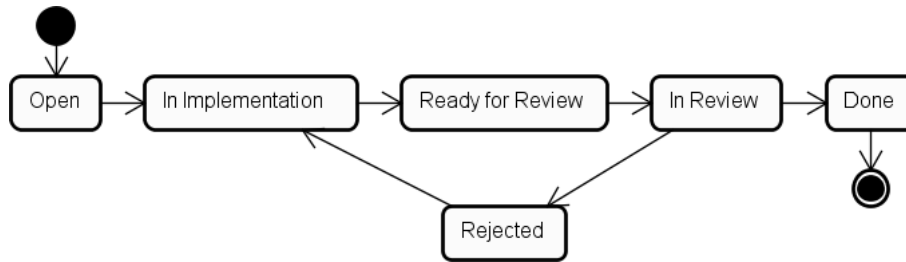
Figure 8.1: Basic state diagram for tickets (Source: [34])

example, all developers use a common setup of the Eclipse IDE.

The company introduced regular code reviews in 2010, and the general process stayed stable over the last years. For almost every development task (ticket), a review is performed for the corresponding code changes. When there are review remarks, these are usually addressed in further commits for the same ticket. In the following, the review process is described along the facets from the classification scheme that is introduced in Chapter 6:

Unit of work = Task: The team divides user stories into separate implementation tasks. A review is performed for each such task. Besides, there are "bug" and "impediment" (i.e., internal improvement) change tasks that are reviewed, too. All change tasks are hosted as tickets in Atlassian Jira.

Enforcement for triggering = Tool: Separate states in the Jira ticket workflow ensure that a review candidate is created for each task. The relevant states are "ready for review", "in review", "rejected" and "done". Figure 8.1 shows the basic state diagram for a ticket.

Publicness of the reviewed code = Post-commit: A review is performed after the changes are visible for other developers, i.e., after committing to the Subversion repository.

Means to keep unreviewed changes from customer releases = Organizational: Before a release, reviews have higher priority and it is checked that all critical reviews are completed.

Means to ensure swift review completion = Priority: Open reviews have to be completed before new work is started.

Blocking of process = Full Follow-Up: A ticket is not considered done before all changes have been accepted by a reviewer.

Number of reviewers / Rules for reviewer count: There is usually one reviewer per ticket, but for specific modules two reviewers are mandatory. Pair programming reduces the number of needed reviewers by one.

Reviewer population = Everybody: Every team member shall be available as a reviewer for every change, albeit very inexperienced team members usually don't review alone.

Assignment of reviewers to reviews = Pull: A reviewer chooses among the open reviews.

Tool support for reviewer assignment = No support: There is no decision support for the choice between the open reviews.

Interaction while checking = On-Demand: The review participants only interact on-demand, e.g., when there are questions regarding the code, and mostly asynchronously. Very rarely, reviews are performed in an in-person meeting.

Temporal arrangement of reviewers = Sequential: Only one reviewer reviews a ticket at a time and rework is done after each reviewer.

Specialized roles = No roles: Even if there are multiple reviewers, they do not take on distinct roles.

Detection aids = Sometimes testing: Sometimes, reviewers perform a limited amount of manual exploratory testing during reviews.

Reviewer changes code = Sometimes: The reviewers may change code during checking and commonly do so for minor changes.

Communication of issues = Written: The found issues are mainly communicated in writing and stored in Jira.

Options to handle issues = Resolve, Reject: The author usually fixes observed issues right away or decides together with the reviewer that the remark will not be fixed. Consequently, review changes are almost always done in the same ticket as the original implementation.

Tool specialization = General-purpose, later Specialized: The traditional way of performing reviews in the company is by looking at the source code changes in TortoiseSVN, a graphical client for Subversion. In 2016 CoRT was introduced as a specialized review tool that could optionally be used by the developers.

9

The Code Review Tool 'CoRT'

There are two alternatives for the review tool platform to use in the partner company: Build upon an existing review tool, or implement a specialized tool. Section 9.1 describes why the second option is chosen, based on major requirements for the tool. The created tool is named CoRT. The sections after that describe CoRT from the perspectives of user (Section 9.2), researcher (Section 9.3), and tool developer (Section 9.4).

9.1 Key Design Decisions

This section motivates key design decisions for CoRT: Why not build on an existing tool? What main features are needed? Why extend the IDE?

The decision to not build on an existing tool is based on two main reasons: Existing tools did not fit the partner company context well, and they were not easily extensible for research. The analysis of existing review tools in Section 7.3 shows that they lack one or several features needed in the context of the partner company: Support for post-commit reviews, integration with Subversion, and streamlined Jira integration. It might have been possible to change the company's environment to accomodate for some of the limitations of the tools, but this would have meant an invasive, big bang approach. By building a tool that fits well into the context, it is instead possible to migrate to the new review tooling step-by-step. No developer is forced to use CoRT.

Another drawback of the existing tools is that they are hard to extend with research features. One is either limited to the public APIs, has to change the source directly, which is only possible if it is available, or resort to hacks like changing the created HTML with browser plugins. A further feature that is deemed important based on the interviews in Part I but missing from most existing tools are IDE-like code navigation and support features.

The basic features of a review tool, like connecting to the ticket system, getting the code changes and highlighting them in the IDE, are not hard to implement building upon open-source libraries. Still, possible drawbacks of this decision are that there is a risk of spending too much work on non-research implementation, and that there is no existing user base, meaning less empirical data and less relevance of the development. The latter drawback is considered

acceptable due to the focus on reviews in the partner company and in-depth research in its context.

The core feature requirements for the tool were extracted from the analysis of existing tools and the results of the interviews. The tool needs to manage the review process, determine the changes that need to be reviewed, allow viewing these changes, and provide facilities to collect, store and distribute review remarks. It is also found that cross-cutting requirements like good performance and usability are important.

The interviews from Part I clearly show that developers desire IDE-like navigation features for reviews. Still, many existing review tools are browser-based. It is certainly possible to implement such features in a browser-based tool, with Upsource being a prime example. But integrating the review tool into the IDE is a much easier way to gain these features. Specifically, the partner company consistently uses the Eclipse IDE, so CoRT is implemented as an Eclipse plugin. A drawback of this decision is that the tool is restricted to teams that use this IDE. Although many parts of CoRT do not rely on Eclipse (see Section 9.4), adapting CoRT to a different IDE is not as easy as adapting it to a different SCM or ticket system.

9.2 CoRT from the User's Perspective

This section gives on overview on CoRT's use and its main user interface components. Based on the result of the interviews in Part I and the research by Myers et al. [278], it is a major concern to provide a streamlined review experience with good usability. The user does not have to focus on the minutiae of the bookkeeping, but can focus on understanding and checking the source code.

CoRT builds on several of Eclipse's standard UI mechanisms. One of these mechanisms is that there are distinct 'perspectives' that can be customized for specific tasks, like programming or debugging. CoRT adds two such perspectives, one for reviewing and one for fixing review remarks. The fixing perspective contains two additional views, one with information on the ticket and one with the review remarks. It is not discussed further in this thesis.

An example of the review perspective can be seen in Figure 9.1. CoRT provides several views for the review perspective. The 'review info' view (Point 1 in Figure 9.1) shows general information on the review and the reviewed ticket. It also links to the ticket itself, as a simple means to implement the *Include Documents* feature from Section 7.1. As CoRT is integrated in the Eclipse IDE, all of the IDE's editors and views can be used during review. Point 2 in Figure 9.1 shows how a portion of the code that needs to be reviewed is highlighted with a violet background in the standard Java editor. Such a code portion is called 'review stop' in CoRT, based on the tour-stop metaphor used in other publications [285, 337]. The 'review content' view (Point 3 in Figure 9.1) shows the hierachical tours and stops that need to be reviewed (see Chapter 14), along with their classifications (see Chapter 15). This view also shows which of the stops have already been reviewed and for how long, by using different shades of green on the stop's icons. The reviewer can directly navigate to a stop by clicking on it in the tree, or he or she can navigate to the next non-reviewed stop. CoRT also indicates when all parts of the code change have been reviewed. In this way, CoRT helps to reach the *Thoroughness* demanded in Section 7.1 without burdening the reviewer with keeping track of what was reviewed.

Reviewing in the most recent version of the code, as shown in Point 2 in Figure 9.1, can be argued to be beneficial because the most recent version of the code is also the one that will be deployed. Furthermore, it allows integration with standard IDE features like code cross-
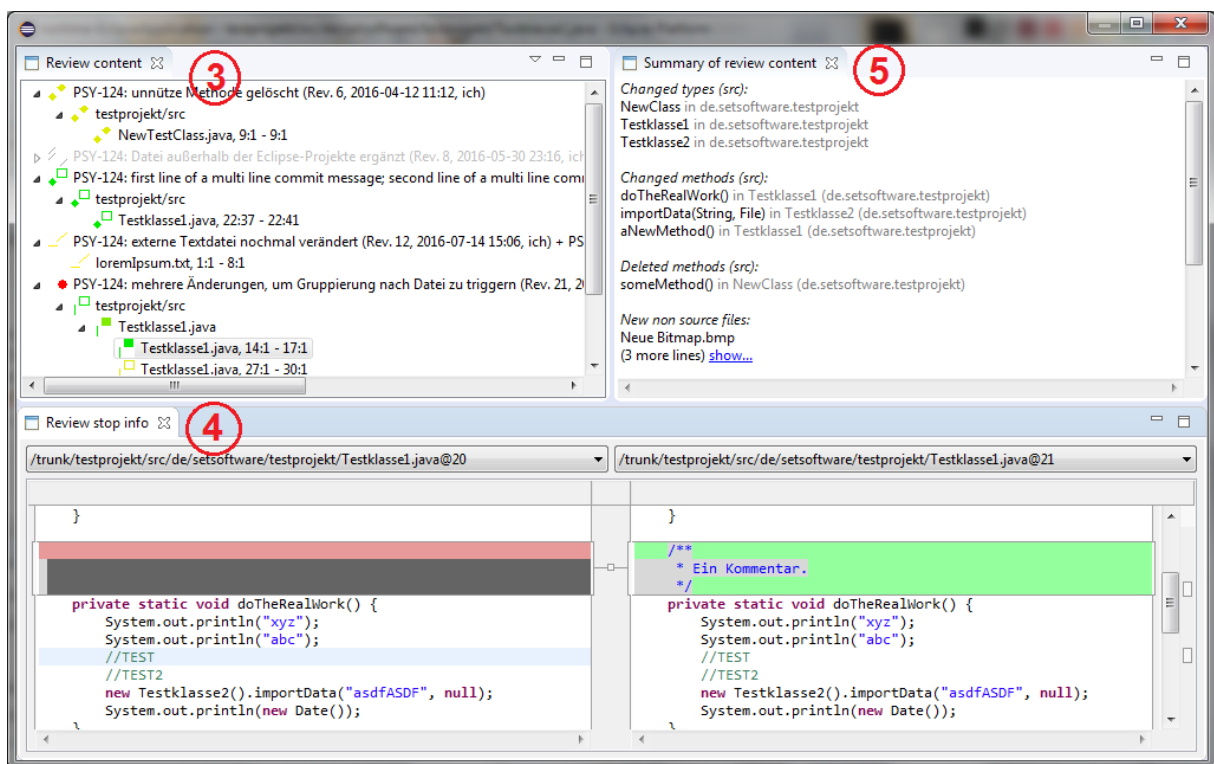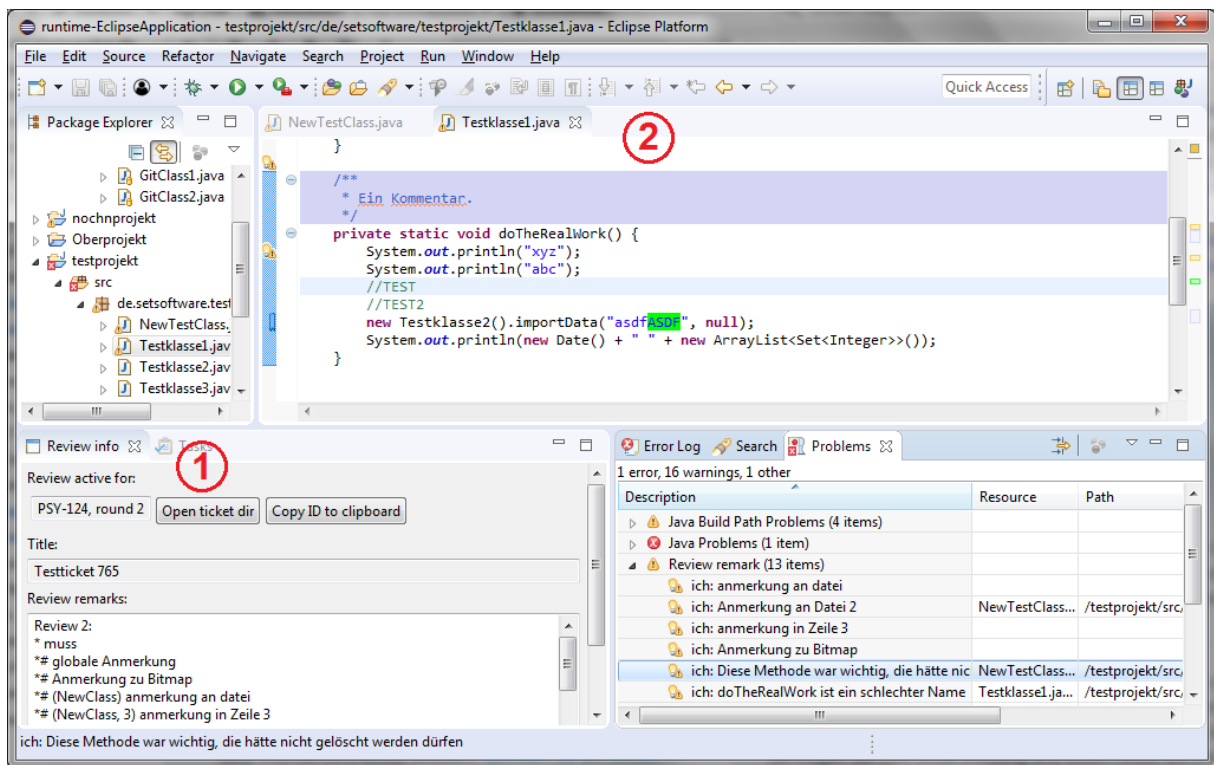
Figure 9.1: Exemplary screenshot of CoRT's review views. The example is based on a developer working with two screens (upper and lower half of the figure).
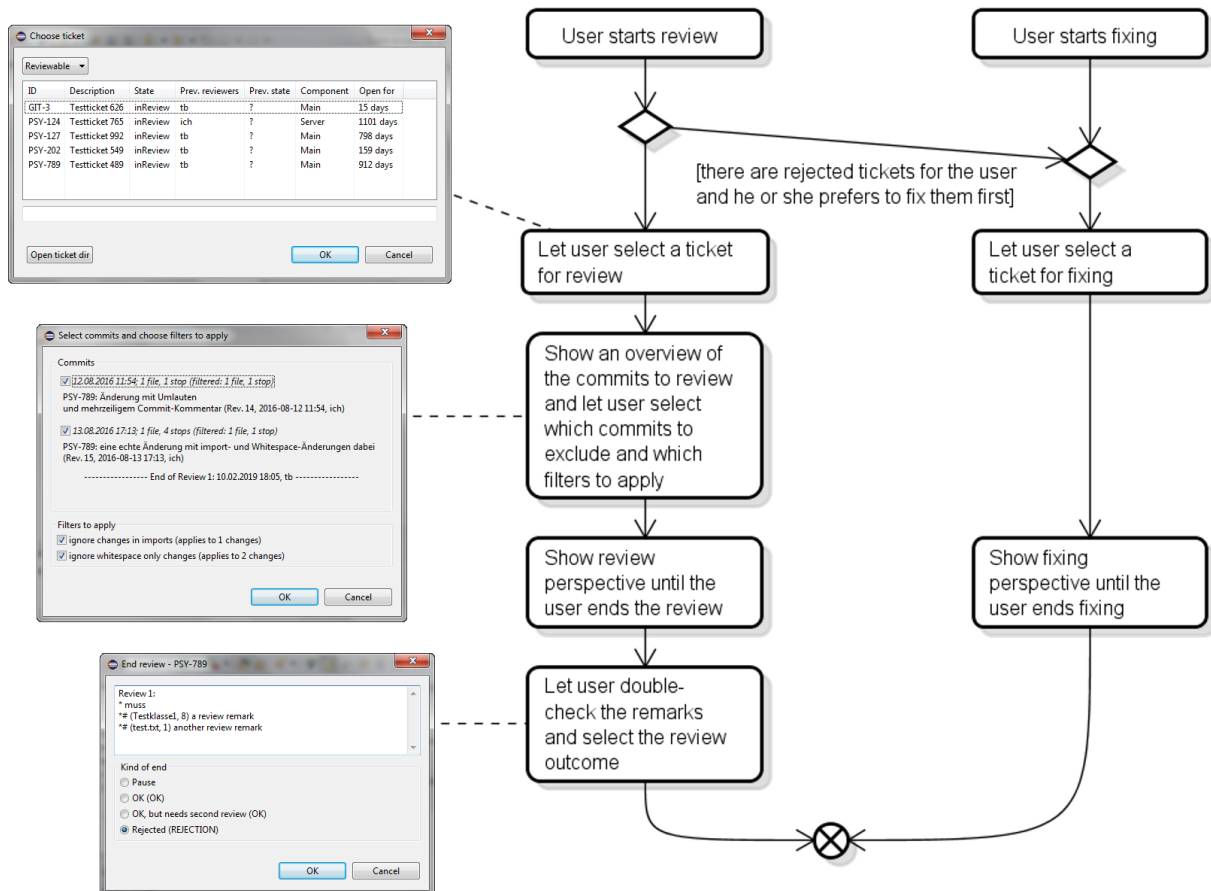
Figure 9.2: Interaction flow and main dialogs

referencing. But sometimes it is much easier to understand and verify a code change by looking not only at the new state, but at the diff of the change. For these cases, the 'review stop info' view (Point 4 in Figure 9.1) shows a diff view of the stop. The final view shown in Figure 9.1 is the 'summary of review content' view that provides a textual summary of the changes to be reviewed, which allows the reviewer to get an overview of the code change. This view was implemented by Roman Gripp in his master thesis [150].

The main checking is done in the review perspective, but several other dialogs interact to enable a streamlined review. Figure 9.2 shows an overview of the main interaction flow and dialogs. The top-most dialog in the figure allows the user to select the ticket he or she wants to review. After selecting a ticket, the respective commits are analyzed and CoRT shows an overview of the commits. This dialog also displays when earlier reviews of the ticket happened, and which of the stops CoRT would filter out (see Chapter 15). The user can adjust the selection of commits and filters. At the end of the review, the user selects which transition in the ticket workflow shall be used (for example 'Review OK' or 'Reject').

As shown in Chapter 4, a characteristic of regular, change-based code review is that subjective planning decisions have been replaced by team-wide rules and conventions. This demands a team-wide configuration of most of CoRT's settings. This configuration is stored in an XML file that can be checked in to version control. Other settings, like passwords or cognitive preferences, can be customized by each user based on Eclipse's standard configuration mechanisms.
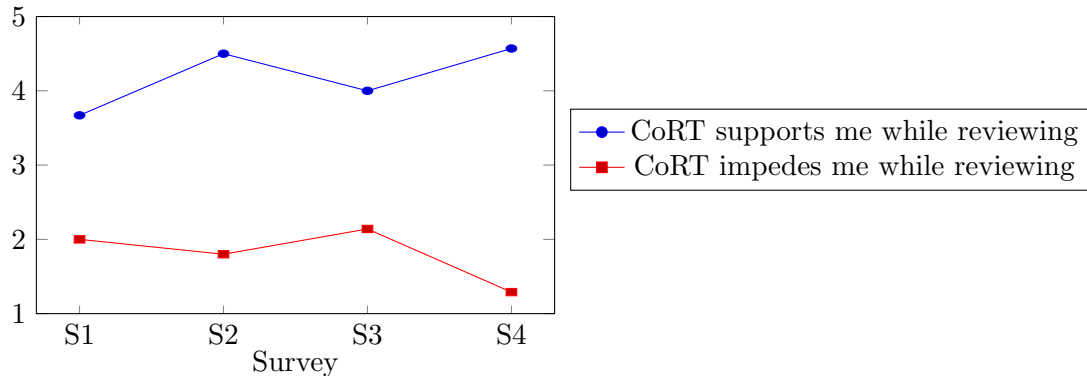
Figure 9.3: Mean ratings from longitudinal CoRT user surveys. Both results are from a 5-point scale, from 1 (don't agree at all) to 5 (totally agree).

Informal user feedback was gathered continuously since CoRT was deployed in the partner company. Apart from that, I handed out brief surveys to gather anonymous qualitative and quantitative feedback at four points in time. The three first surveys were done at the start of the deployment, each spaced about half a year apart, and the last survey was done near the end of the thesis timeframe two years later. Figure 9.3 shows the results from the questions on CoRT's utility. Having two questions might seem redundant, but apart from avoiding bias, this also helped to stimulate responses both on things that are liked a lot and that need to be improved. Among the positive things that cropped up most often are: The streamlined usability and integration with the development context, the classification of review remarks (see Chapter 15) and the tracking of reviewed portions of the change. Negative comments often revolved around performance problems, the early diff views (see Chapter 11), and badly supported contexts, e.g., for programming languages that are not well supported by the installed Eclipse IDE.

9.3 CoRT as a Research Platform

The CoRT code review tool has two goals: To be a practically usable code review tool, and to be a platform for research on better code review tooling. To act as a research platform, CoRT includes software telemetry features, and it is extensible in several ways.

Software telemetry is an approach that uses instrumentation of software development tools to collect metrics automatically and unobstrusively [412]. It was pioneered for review tools by Johnson [185], with other contributions, e.g., by Halling et al. [154]. CoRT, too, is instrumented to log various events on its use, for example when a review is started, the viewed file is changed, code is executed, and many more. Each event contains a timestamp, a pseudonymized developer ID, an identifier for the review session, the event type, and several fields depending on the type. These events are centrally collected, and the collected data can then be processed further and analyzed. The developers at the partner company could decide whether to activate CoRT's telemetry features. The collected data is used, for example, in the ordering study of Chapter 14.

The 'Hackystat' project at the University of Hawaii was a major milestone for software project telemetry. To stay compatible to potentially existing tools of other researchers, CoRT uses the same XML format as Hackystat [412]. A lightweight library that logs the events as files on a shared network directory is a part of CoRT.

The second aspect that increases CoRT's utility as a research platform is its extensibility.
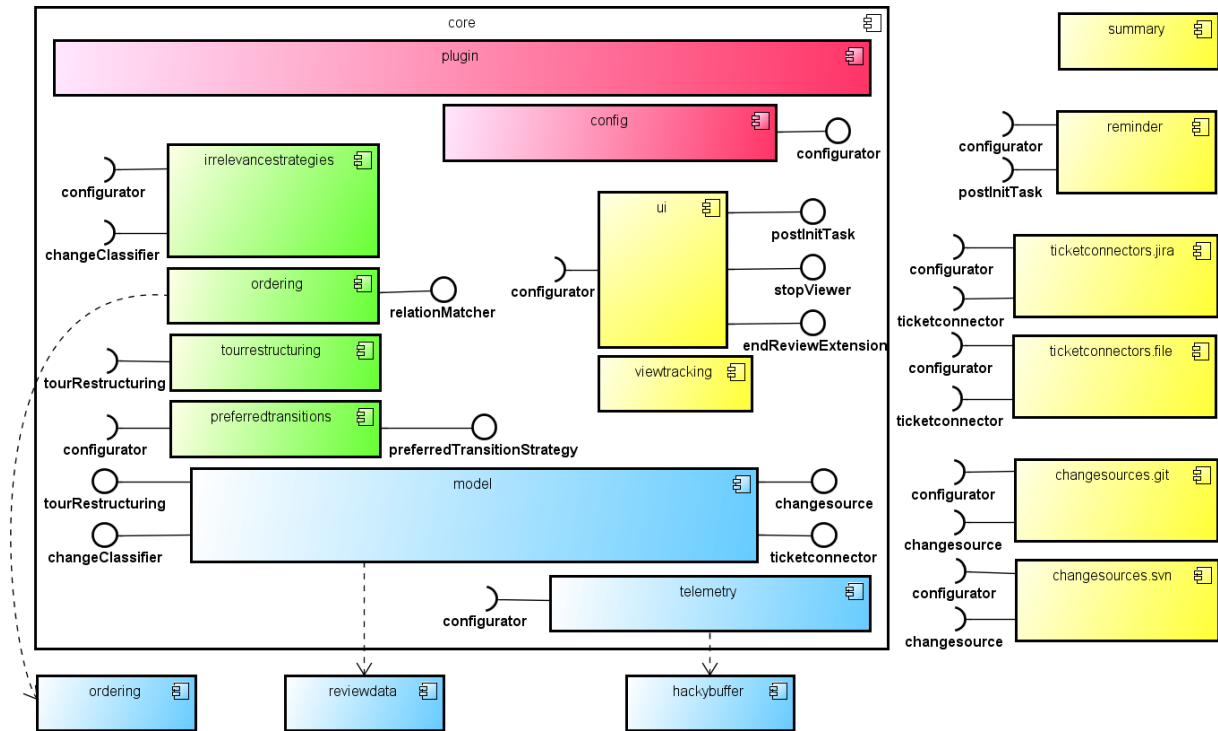
Figure 9.4: Component diagram for CoRT. To reduce clutter, dependencies inside the 'core' plugin are not shown.

As a part of the Eclipse IDE, other Eclipse plugins can naturally be used in combination with CoRT. But more specific extensions are possible, too: CoRT provides an OSGi extension point for so called 'configurators'. Plugins that implement this extension point can be configured together with CoRT, and use this lifecycle to hook into further parts of CoRT's API.

9.4 Overview of CoRT's Internal Architecture

The following section describes the basic structure of CoRT's source code and some key mechanisms in its implementation. Figure 9.4 shows an overview of CoRT's architecture. Each top-level component is an OSGi plugin. For the 'core' plugin, the figure also shows the internal package structure. Coloring is used to classify the components (inspired by Quasar [346]): Common base components are shown in blue, algorithms for cognitive-support review tools that are mostly independent from the development context are green, components that interface to the user or to external systems are yellow, and glue code that assembles the components into a working application is red. The blue and green parts are largely decoupled from the Eclipse IDE. The responsibilites of the components are:

core.plugin is responsible for the lifecycle of the Eclipse plugin and for integrating the other components into a working application.

core.config provides a mechanism for other components to be configurable based on a single team-wide configuration.

core.ui is a large component with several sub-packages and provides the views, dialogs and other user interface implementations of CoRT.

64

core.viewtracking is responsible for tracking which parts of the code change have already been visited, based on the user's interaction with the IDE.

summary is a mostly independent plugin that provides the textual change summary view developed in Gripp's master thesis [150].

reminder is a largely independent plugin that informs the user when there are too many tickets waiting for review, or when they are waiting for too long.

ticketconnectors.jira connects CoRT to the Jira ticket system to store review remarks, show open reviews and interact with the ticket lifecycle.

ticketconnectors.file provides a simple file-based storage of tickets and review remarks, mostly for testing.

changesources.git provides the ability to extract code changes from the git SCM.

changesources.svn provides the ability to extract code changes from the Subversion SCM.

core.irrelevancestrategies implements several strategies to classify code changes based on their relevance for review. Further details on this topic are described in Chapter 15.

core.ordering implements several strategies to determine relations between change parts and to find a good order of reviewing them. This topic is discussed in detail in Chapter 14.

core.tourrestructuring provides strategies to combine the parts of the code change under review into stops and tours. This is especially relevant when the same code portion was changed in several commits.

core.preferredtransitions provides strategies that decide which 'transition' in the ticket workflow shall be pre-selected in the 'End review' dialog, based on the found remarks and the contents of the review.

core.model implements the domain model and provides interfaces to customize parts of the review process. Important parts of the domain model are the 'changestructure', dealing with commits and the contained changes, and the tours and stops that are later created based on the changes.

core.telemetry provides the telemetry functionality that is described in Section 9.3.

hackybuffer is the light-weight implementation of Hackystat's XML format that is mentioned in Section 9.3.

ordering implements the efficient ordering algorithm of Appendix D, independent from the review domain.

reviewdata contains the domain model for review remarks, which is described below, and allows serialization and deserialization of these remarks in textual form.

Figure 9.5 shows the model for review remarks that is used in CoRT, i.e., the model underlying the 'reviewdata' component. It is based on the review practices of the partner company, and on further input gathered from the interviews of Part I. With a sequential temporal arrangement of reviewers (see Section B.3), the overall review data is split into several review rounds. In each round, the reviewer(s) can add review remarks. These review remarks contain an initial textual comment, and are bound to a certain position in the code base. Positions can point to a specific line in a file (in the reviewed revision), but sometimes it is more adequate to consider a remark to apply to a file as a whole, or even to the whole ticket ('GlobalPosition'). A remark can spark further discussion, which is modeled as a simple thread of additional comments. To mark a remark as considered, the author can add a resolution to it, either stating that he or she fixed the problem, does not agree with the remark ('Rejected') or needs further clarification ('Question'). Each remark is classified by the reviewer into one of several types: For remarks that demand action ('ToFix'), the reviewer can specify whether he or she regards addressing it as mandatory ('MustFix') or just as a suggestion ('CanFix'). Furthermore, there are remark types that do not
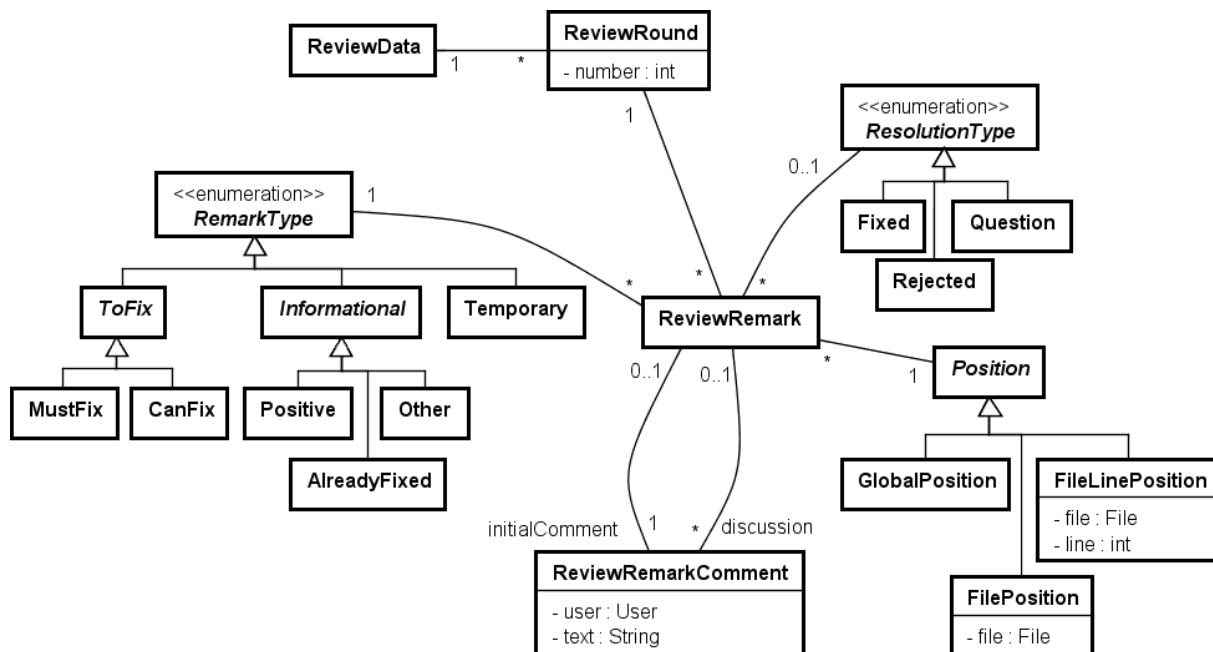
Figure 9.5: Domain model for review data

demand an action from the author and just serve to convey information: Remarks that were 'AlreadyFixed' on-the-fly by the reviewer, remarks about positive aspects, and other remarks, for example on process issues. The remark type 'Temporary' is special in that it is only used during the review to store ideas that the reviewer wants to follow-up later during the review. In this way, they help the reviewer to off-load mental load (see Chapter 12), and they could also be used to store the results of static analyzer or other agent that should help the reviewer (see Chapter 15).

$$\diamondsuit$$

This chapter could only provide a brief overview of CoRT. It shows its utility from the user's perspective, reflected in very positive ratings, and also its features as a research platform and how it is designed to be reusable and extendable by other researchers. CoRT's full source code is available online.[1] The central findings gathered while designing CoRT are summarized in abstract form in Appendix A. In addition, the next two chapters describe two studies done to inform the design of CoRT, before Part III goes into details on its cognitive support features.

---

[1]https://github.com/tobiasbaum/reviewtool, licence: EPL

10

A Simulation-Based Comparison of Pre- and Post-Commit Reviews

In Chapter 6 it is shown that development teams are split between two process variants, depicted in Figure 10.1: Some perform reviews after the code change has been integrated into the main development branch (post-commit review), whereas others review before integration (pre-commit review). This process choice has a large influence on a review tool such as CoRT, and supporting both would be effortful. The partner company traditionally used a post-commit review process. In recent scientific publications, pre-commit reviews are more prevalent, especially in the form of pull requests. Therefore, a simulation study is performed to show under which circumstances pre- or post-commit review is better, and whether the partner company should change its process. This chapter describes the basics of the study, its results, and its limitations. Details on the simulation model are provided in Appendix C. This chapter uses material from [34] and [35], joint work with Fabian Kortum, Kurt Schneider, Arthur Brack, and Jens Schauder.

In the interviews for the Grounded Theory study in Part I, the developers named several reasons why they believe that either pre- or post-commit reviews are more efficient:

- Pre-commit reviews find defects before they impede other developers.
- Pre-commit reviews might extend the cycle time of user stories, increasing the 'work in progress' and consequently increasing task switch overhead. On this basis, Czerwonka et
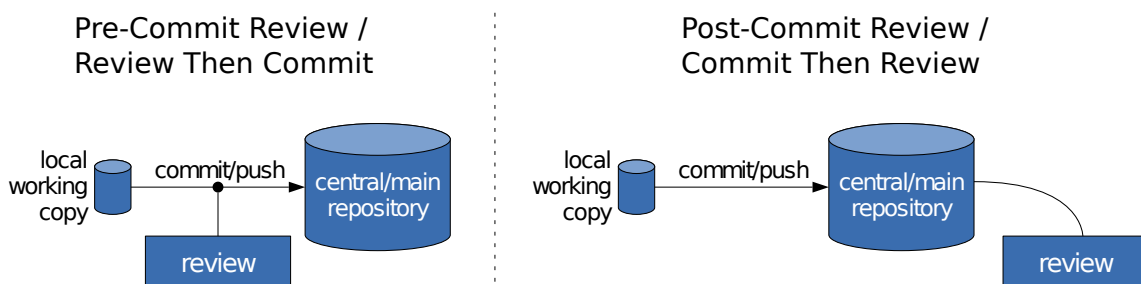
Figure 10.1: Different levels of publicness of the reviewed change: Pre-commit review and post-commit review

67

al. [85] criticize that "*code reviews [...] often do not find functionality defects that should block a code submission.*"

- Post-commit reviews support the early and often integration of changes, better suiting the mindset behind continuous integration and possibly reducing the risk of conflicts.

This chapter assumes that the best choice might depend on the context and that some of these effects have a larger influence than others. Consequently, the research questions are:

RQ$_{10.1}$**.** Are there practically relevant performance differences between pre-commit and post-commit reviews?

RQ$_{10.2}$**.** How are these differences influenced by contextual factors?

As these questions are still quite broad, they are further detailed in Section 10.1. The study shall result in a better understanding of the effects responsible for performance differences. This knowledge can then be used by the partner company and others to decide which variant is adequate for their situation. To ease adoption, the knowledge is distilled into a set of simple, practically usable heuristics.

A simulation-based approach is used to assess the research questions, instead of a controlled experiment or a number of case studies. Performing a simulation permits to study a vast number of different situations in a holistic way and with less effort than a large scale experiment.

In terms of related work, no other simulation studies explicitly target 'modern' change-based code review, but many studies used simulation to examine classic forms of inspection and review: Among these are system dynamics models, like the ones by Madachy [242] and Tvedt and Collofello [381], as well as discrete event models, for example by Neu et al. [281] and Rus, Halling and Biffl [329]. The requirements inspection process has been studied by Münch and Armbrust [276], as well as Wakeland, Martin and Raffo [391].

Simulations have been used to assess other questions in the context of agile software development: Turnu et al. [380] used a system dynamics model to investigate the effect of test-driven development in open source development, and Melis et al. [262] built a model to analyze the effects of test-driven development and pair programming. Anderson et al. [10] used simulation to compare Scrum, Kanban and a classic process for a team at Microsoft. A comprehensive overview of further software process simulation studies can be gained from the systematic literature reviews done by Zhang, Kitchenham and Pfahl [411] and Ali, Petersen and Wohlin [8].

10.1 Methodology

From a very high-level view, the 'in silico' study described in this chapter consists of two iterations of the scientific cycle of induction and deduction: I start with mostly qualitative empirical observations, create a simulation model (i.e., an executable theory) based on these observations, use this model to deduce quantitative data, which can then be used to validate the model but also as the starting point of another cycle, inducing heuristics and checking them against the simulation results. A closer look reveals that each cycle, in fact, consists of several iterations of refinement and validation.

10.1.1 Performance Metrics

The term "performance difference" from this chapter's general research question is too vague to be used directly. Following the goal question metric paradigm, it is split into three questions corresponding to the classic target dimensions of software projects: quality, cost/efficiency and time to market. The chosen metric for quality is 'number of issues found by customers per story point'[1]. Efficiency is measured as 'number of story points completed' because the actual effort (i.e., number of developers times simulated duration) is fixed for a given situation. Time to market is measured as 'mean user story cycle time', i.e. the average time from the start of a story to its delivery to the customer. The adequacy of these measures has been checked in a questionnaire answered by twelve industrial software developers, a convenience sample taken from the local Java User Group. The respondents also gave bounds for the differences that they regard as practically relevant in the context of the study. The resulting intervals (and order of importance) are $\pm7.5\%$ for quality, $\pm10\%$ for efficiency and $\pm15\%$ for cycle time.

10.1.2 Iterative Creation of the Model

I used the model development process described by Ali and Petersen [7], in addition to the guidelines from Law and Kelton [228] and Page and Kreuzer [291] to create the simulation model.

The conceptual model is based on review processes observed in practice. These observations mainly stem from the interviews on the state of the practice described in Part I. The model was reviewed and checked for face validity by two experienced practitioners, one using pre-commit reviews and the other using post-commit reviews in his daily work. Both of these practitioners are lead software developers and have worked at several different companies. One has about eleven years of experience in industrial software development, the other about 19 years. The reviews of the conceptual model were conducted as a walk-through [179]. Further validation measures are described in Section 10.3.

The model is implemented as a discrete event simulation model using the DESMO-J simulation framework [291]. Verification is done with a combination of several methods. Among those are software engineering best practices like unit tests and code reviews. Additionally, a group of three researchers (Baum, Kortum, Kiesling) checked parts of a simulation trace in detail by enacting it. Several iterations of global variance-based sensitivity analysis were performed using the method of Sobol' [332]. Parameters that were shown to be of little influence were set to fixed values. After several iterations for verification, I started to generate data to analyze the main effects and derive heuristics. Details for this sampling process are described in Section 10.2.2.

10.1.3 Data Generation and Analysis

The model is parametric, with input parameters for the details of the development context, for example, how skilled the reviewers are and how long it takes for customers to find injected defects. Details for the parameters are described in Appendix C. For a given situation, the model is executed both using pre-commit and post-commit reviews, and the relative difference for the three output metrics is calculated. It is also executed without reviews, to be able to exclude data points for which any type of code review is not advisable. The execution is repeated with different random number seeds (using common random numbers [228] for variance reduction) to

---

[1]I first used 'number of issues found by customers', but this is highly correlated with the amount of work finished.

obtain a median difference and its confidence interval. Some more details on the data generation are described in Section 10.2.2.

To extract the heuristics, I used an iterative process based on exploratory data analysis and local sensitivity analysis: I formed working hypotheses, mainly by looking for correlations and other distinctive patterns in scatter plots of subsets of the simulation results. These hypotheses were then checked by two types of local sensitivity analysis [332]: For a single data point, one factor at a time was increased and decreased, and two data points with opposing results were systematically interleaved. This process led to an increased understanding of the model's main effects but was also quite effective as a further method of verification.

10.2 Results

To derive simple heuristics for the use in practice, the study analyzes a broad range of different contextual factors. These results are shown in Section 10.2.2, but beforehand a specific data point based on the partner company is analyzed.

10.2.1 Details for a Specific Data Point

The example in this section is based on data for the partner company. Some parameter values were derived from the company's ticket system and the rest is based on expert judgment. All these values have to be taken with a grain of salt: The ticket system data likely contains systematic biases, among others due to a tendency to forget to change a ticket's state or to pause working on a task without changing its state. The expert estimates are impeded by a lack of intuition for some of the parameters. The detailed values can be found in Appendix C.

Figure 10.2 shows an output from a simulation run with these estimates. It shows simulation time on the x-axis and the number of started and finished user stories as well as the number of issues observed by the customer on the y-axis. The sharp drop in the diagram marks the end of the warm-up period of 700 days. All counters were reset at that point. Only data collected in the 600 working days (about 3 years) after that is used for further analysis. At the very start of the simulation, the number of issues found by customers increases slowly due to the associated delays in the model. After that, all values show a roughly linear rise, as expected for a continuous Kanban process. It can be seen that for this data point, the warm-up period is by far large enough. It is sized this large to allow the same value to be used for data points which need a longer warm-up.

Table 10.1 shows the results after 42 simulation runs with different random seeds. For each output dimension, the table shows the median relative difference, the median's 99% confidence interval and the minimum and maximum relative difference obtained. Efficiency is measured in

Table 10.1: Relative differences for the example data. A positive relative difference means that post-commit reviews have the larger value. For efficiency, the larger value is better, for quality and cycle time the smaller. All percentages have been rounded to the nearest integer for presentation.

	Median	Tendency	Confidence interval	Minimum	Maximum
Efficiency	0%	–	-1% .. 1%	-3%	6%
Quality	-2%	post	-5% .. 1%	-11%	10%
Cycle Time	-12%	post	-15% .. -11%	-22%	-6%
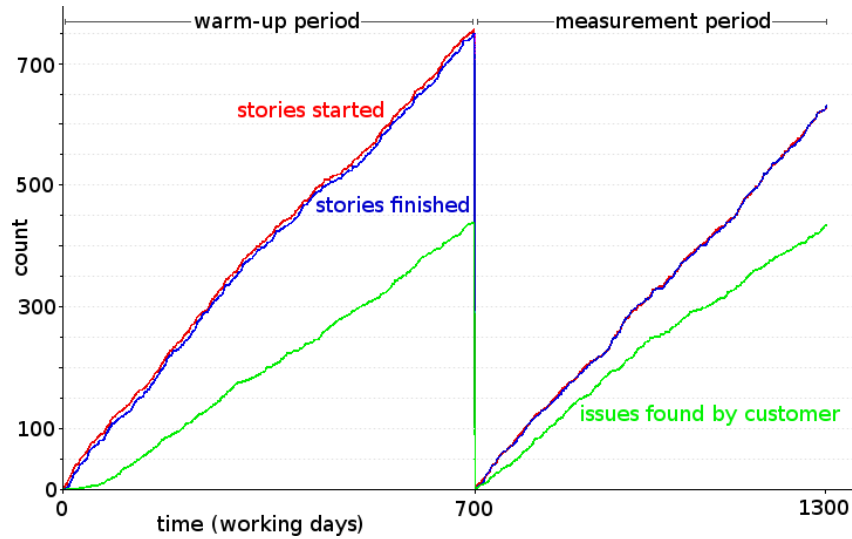
Figure 10.2: Example output from a simulation run: Stories and issues over time (Source: [34])

terms of 'finished story points', quality in 'issues found by customers per story point' and cycle time by the 'mean cycle time of a user story'. Those results show that the difference in terms of efficiency and quality is very small for this data point. The difference in cycle time is much larger, but still not practically relevant, according to the limits for practical relevance gained in the respective survey (-15% to 15%).

10.2.2 General Results

To understand the influence of different contextual factors and to derive heuristics, a total of 289,230 random data points was sampled in the final analysis run. This was done after several iterations of model verification, refinement and parameter exclusion through global sensitivity analysis. To obtain a data point, each input value was sampled uniformly from an interval (see Tables C.1 and C.2 in Appendix C for the detailed intervals). These intervals were estimated jointly by two researchers and two practitioners (one a developer from the partner company), with a tendency to "if in doubt, enlarge". For each sampled point, the simulation repeatedly ran with different random number seeds at least 20 times. This process continued until a statistically significant classification as one of 'pre better', 'post better' or 'negligible (not practically relevant) difference' for each of the three target attributes was possible. Furthermore, it was checked that the data point is not obviously unrealistic (more than 80% of the invested effort wasted on fixing etc.) and that performing code reviews is beneficial at all. To err on the safe side, the number of finished story points without review had to be higher than 110% compared to the next best alternative to dismiss review. After excluding these cases, as well as those where statistical significance was not reached after 2,000 repetitions, 50,901 data points remained. By far the largest number of exclusions were situations in which almost no work was finished due to high error rates and ineffective fixing. Based on these results and the combination of manual analysis and local sensitivity analysis described in Section 10.1.3, I identified the main effects. They are described in the following.

10.2.2.1 Analysis for Quality

Regarding the number of issues found by customers per story point, there is one main effect that leads to a practically relevant difference between pre- and post-commit reviews: Soon after a change is committed, other developers can spot (or 'stumble upon') issues. The time between this point and the delivery of the story is generally larger for post-commit than for pre-commit reviews so that more issues are found outside of reviews. This effect is larger the higher the chance that developers will really do so, e. g. they are really good at finding issues or there are many of them, and the effect is larger the higher the difference in time between post-commit and pre-commit reviews. The relative difference is also higher for lower review skills. That being said, the difference is not practically relevant for the vast majority of the sampled situations. The only effect I found that leads to a preference for pre-commit reviews in terms of quality is an edge case: A high number of global blocker issues[2] leads to more review rounds for pre-commit reviews, and therefore increases total review effectiveness compared to post-commit reviews.

10.2.2.2 Analysis for Efficiency

For differences in efficiency, there are more effects interacting: As described for quality, post-commit reviews lead to more issues found by uninvolved developers. When review skills are low, this is a chance to find more issues while the author is still working at the task, which is beneficial for efficiency. With higher reviewer effectiveness, when most issues would be found in a review anyway, letting other developers find them is detrimental to efficiency. The relation between task switch overhead and review mode is double-edged, too: For small teams with task dependencies, task switch overhead is smaller for post-commit reviews, but for larger teams, the task switch overhead for issues spotted by other developers grows and makes pre-commit better. Another relevant effect that is beneficial for pre-commit reviews stems from global blocker issues: The effect of blocker issues increases with a growing risk of injecting them, increasing time to recover from them and a growing number of affected developers. It also rises with the probability of finding blocker issues during a review.

10.2.2.3 Analysis for Cycle Time

The largest difference between pre- and post-commit reviews exists in terms of cycle time: When there are dependencies between tasks so that one task has to be committed before another can be started, this leads to longer cycle times for pre-commit reviews. Figure 10.3 shows this quite well. The relative effect becomes larger when the time between initial implementation and the end of the review cycle grows in comparison to the total cycle time (e.g., through long review times or a large number of review rounds). This effect is so large that post-commit review has a smaller cycle time in the majority of sampled cases with dependencies, whereas without dependencies a negligible difference in cycle times is prevalent. A second effect that affects cycle time is again the same as that responsible for differences in quality: With post-commit reviews, more issues can be found by developers other than the reviewers. This usually takes more time than finding and fixing them as review remarks. When it does not, and review effectiveness is low so that a significant number of issues will pop up after review, this can also lead to lower cycle times for post-commit review. The 'global blocker' effect described for efficiency is relevant for cycle time, too.

---

[2]'global blocker issues' are issues that will block many developers as soon as they are integrated, e.g., if someones commits a compile error; see Appendix C
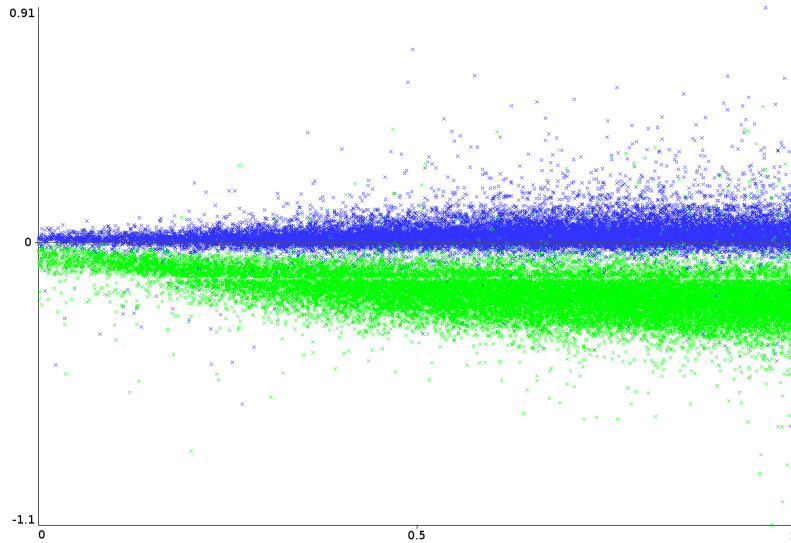
Figure 10.3: Scatter plot of reviewer effectiveness (x-axis), relative difference in cycle time (y-axis) and dependency graph constellation (color; light/green = REALISTIC, dark/blue = NO_DEPENDENCIES; see Appendix C for a description of the dependency structures) (Source: [34])

RQ$_{10.1}$: *Under certain circumstances, there are differences in quality, efficiency or cycle time between teams that use pre- or post-commit reviews. The most common difference is a higher cycle time for pre-commit reviews, the other differences are rare.*

10.2.2.4   Resulting heuristics

Figure 10.4 shows the heuristics derived from the findings on the main effects and the simulation results. When formulating heuristics, the question is not if there will be inaccuracies, but rather which inaccuracies are least troublesome. In that regard, the simplicity of the rules and the recall for 'pre better' and 'post better' is valued highest. The accuracy of the heuristics for the simulation results can be seen in the confusion matrices in Table 10.2. They show for each of the three result classes the number of data points classified by the heuristic as belonging to each class.

RQ$_{10.2}$: *The conditions under which pre- or post-commit reviews are preferable can be stated as heuristics based on several contextual factors, like the dependency structure of tasks, the team size and several aspects of developer skills.*

Further boiling down the heuristics, the main implication of the study to practitioners can be summarized as: If the team currently uses code reviews and has no problem with cycle time or developers being held back by issues that would be found in reviews, it is probably not worth the effort to switch from post to pre or vice versa. If the team does not have an existing process yet, it should use pre-commit reviews if the team is large and cycle time is of little importance or the reviews can be arranged so that no dependent task waits for a review to be finished. It should use post-commit reviews if the team is small or there are dependencies between tasks that

73

For quality:

```
IF veryGoodAtSpottingIssues THEN Post
  ELSE NegligibleDifference
```

For efficiency:

```
IF lowReviewSkill THEN
  IF veryGoodAtSpottingIssues
  THEN Post
  ELSE NegligibleDifference
ELSE
  IF globalIssuesAreAProblemSolvedByReviews OR veryGoodAtSpottingIssues
  THEN Pre
  ELSE IF smallTeam AND dependenciesRelevant AND shortIssueAssessment
  THEN Post
  ELSE NegligibleDifference
```

For cycle time:

```
IF lowReviewSkill OR NOT dependenciesRelevant
THEN NegligibleDifference
ELSE Post
```

Helper definitions:

```
veryGoodAtSpottingIssues := issue activation time mean for developers < 75 days
lowReviewSkill := review effectiveness < 25 %
globalIssuesAreAProblemSolvedByReviews :=
  (global blocker issue risk *
  global blocker issue suspend time *
  number of developers *
  review effectiveness) > 1.5 person-hours
smallTeam := number of developers < 8
dependenciesRelevant := dependency graph constellation != NO_DEPENDENCIES
shortIssueAssessment := issue assessment time mean < 1.5 h
```

Figure 10.4: Heuristics derived from the simulation results. 'issue activation time mean for developers' roughly corresponds to the time it takes a developer to notice a defect or other issue; definitions of this and the other parameters can be found in Tables C.1 and C.2 in Appendix C

Table 10.2: Confusion matrices for the heuristics. For quality, only data points with at least 10 issues found by customers per year were used, because for lower values the "issues per story point" metric is dominated by differences in story points.

Heuristic for quality:

	pre	*post*	*negl.*
pre	0	4	271
post	0	10043	263
negl.	0	2910	16693
classified as →	*pre*	*post*	*negl.*

Heuristic for efficiency:

	pre	*post*	*negl.*
pre	3201	12	1062
post	32	87	44
negl.	9157	1564	35742
classified as →	*pre*	*post*	*negl.*

Heuristic for cycle time:

	pre	*post*	*negl.*
pre	0	39	531
post	0	18738	397
negl.	0	4143	27053
classified as →	*pre*	*post*	*negl.*

would otherwise increase cycle time[3]. For pre-commit reviews in the form of pull requests, one should also keep in mind the benefits that were out of the scope of this study: Pull requests allow easier contribution to open source projects by outsiders [147], and they enforce more process discipline and are an easy way to keep unreviewed changes from being delivered to the customer (see Appendix B.1). Also, what is described as post- and pre-commit review in this study are actually extremes on a scale with multiple mix processes in between. Understanding the main effects can help to find the right mix for a given situation.

10.3 Validity and Limitations

The two parts of the current study, creation of the simulation model and derivation of heuristics, have their own threats to validity. To mitigate these, I mainly used techniques described by Sargent [334] and de França and Travassos [97], in addition to the guidelines referenced in Section 10.1 and general software development best practices.

Several means helped to establish a valid conceptual model: The general structure of the model is based on empirical evidence, gained from the in-depth analysis of the code review processes of several companies in Part I. Two software development practitioners (Schauder, Brack) performed an independent check for face validity. One of them uses pre-commit reviews in his daily work, the other post-commit reviews. In addition, the model was discussed with software developers at a local Java User Group meeting. In contrast, the model's quantitative validation is much more limited, mainly due to missing data: We (Baum, Kortum, Brack, Schauder) estimated some parameter values and fitted distributions based on historical data from the partner company's ticket system. The simulation results of this data point were compared to the real historical outcomes. The results were encouraging, but their value is limited, as many of the model's parameters could be estimated only.

There are additional limitations regarding the quantitative empirical data: It is taken only

---

[3]These heuristics can be derived from the earlier formal heuristics by assuming that: (1) "veryGoodAtSpottingIssues" is false (2) "globalIssuesAreAProblemSolvedByReviews" correlates with team size and is rare compared to differences in cycle time and (3) "NegligibleDifference" can be treated as a "don't care" when optimizing the resulting boolean formulae.

from a single company, and it is based on the analysis of data collected for a different purpose. These limitations do not weaken the main study results, because the values are mainly used to cross-check the results. In the remaining cases, like for determining a realistic task dependency structure, we also ran the simulation model with alternative values. The ensuing sensitivity analysis showed the exact values for these parameters to be of lesser importance.

Like every simulation model, the model is built for a specific purpose (comparison of pre- and post-commit reviews) and contains simplifying assumptions. As these are related to details of the model's implementation, they are described in Section C.4. The computerized model is verified with a combination of techniques: The model was developed iteratively, with a first throw-away prototype followed by incremental refinement. Use of an established simulation framework [291] reduced the chance of defects. In addition to testing, the computerized model's code was reviewed by two persons (Brack, Kortum).

The simulation model is a probabilistic model. Therefore, interpretations are done based on the median and its 99% confidence interval. Following the advice by Pawlikowski et al. [294], the simulation runs were repeated with different random number seeds until the confidence interval was small enough to assign each data point to an output class with statistical significance. All random numbers were drawn from Mersenne Twister pseudo-random number generators.

The developed heuristics intentionally trade accuracy for simplicity. Nevertheless, we took some measures to ensure validity: When deciding whether to include a model parameter in the random sampling space, this was done with a policy of 'if in doubt, leave it in'. Parameter values were only fixed if global sensitivity analysis showed them to be irrelevant. The large random sample of 289,230 data points provides some protection against misinterpretation of random effects. Local sensitivity analysis further ensured that the observed correlations in the output are really based on specific effects.

Regarding external validity, the model is based on a commonly used type of review process, but many real team's processes differ in the details. The main effects discussed for the heuristics can help to decide if the differences between a specific situation and the implemented situation are relevant. The external validity of the heuristics is limited by the fact that the empirical distributions of the input parameters for real development teams are unknown: It is not sure if an erroneous classification for a data point will affect thousands of real teams or no team at all.

$$\Diamond$$

Summing up, this chapter compares two variants of change-based code review, pre-commit review and post-commit review, regarding differences in software development quality, efficiency and cycle time. This is done with a parametric discrete event simulation model that is built based on observations of agile development processes. The model has been validated and verified in a number of ways, including checks for face validity by one lead software developer per variant. The simulation model is used to explore the main effects contributing to performance differences and to derive simple heuristics for the use in practice. Which variant is preferable depends on the context, for example, if task dependencies exist and are relevant for reviews, how fast developers are at spotting issues, and how large the team is. In many situations, there are no practically relevant differences. When there are, there is a tendency for pre-commit review to be better regarding efficiency and for post-commit review regarding cycle time and quality. The results of the study were presented to the development team at the partner company. Based on them the team decided to stay with their process of post-commit reviews. Consequently, CoRT also focuses on post-commit reviews.

11

An Empirical Comparison of Multiple Ways to Present Source Code Diffs

As stated in Chapter 9, CoRT provides two views that show the code to review. One shows the most recent version of the code, whereas the other shows the change parts with both the old and new version. An early release of CoRT contained a view for single change parts, which was changed to the use of a diff view for the whole file according to user feedback. Initially, this view used the standard 'compare view' of Eclipse to show diffs. Before the introduction of CoRT, the developers used the diff view of TortoiseSVN[1], and several developers complained about the new view. The Eclipse view and TortoiseSVN differ mainly in two ways: The use of color, and the alignment of the old and new code in the two-pane diff. Several other available tools also disagree on whether to use color and/or alignment. Therefore, a controlled experiment is conducted to test whether and how these characteristics influence the efficiency of understanding. The current chapter describes the design, execution, results and limitations of this experiment.

Three students, Christian Koetsier, Daniel Schulz und Christian Mergenthaler, designed many of the details of the experiment, as a part of a course on experimentation in Software Engineering. Further empirical data was later gathered by Katja Fuhrmann and Hendrik Leßmann. The author of this thesis co-supervised the students, supported the design and analyzed the resulting data.

There is one other study that analyzes aspects of diff viewers in a controlled experiment: Lanna und Amyot [221] compare two viewers in it, the one by Eclipse and a novel implementation created by them. There are three main differences between these:

- Their implementation uses color.
- Their implementation reduces line jumps for equal content (alignment)
- Their implementation allows interactive switching between old and new version.

Their experiment shows better efficiency with their implementation, but their design confounds the three factors so that the factors' individual influence is unclear. In the context of code review, their interactive switching feature cannot be used well, so that it is not taken into account in this chapter.

---

[1]https://tortoisesvn.net

Figure 11.1: Visualization of the four combinations of color and alignment

11.1 Methodology

The final goal is to show whether the differences in presentation lead to differences for one of the intended review outcomes, mainly for the efficiency and effectiveness of finding issues. But these measures are influenced by many other factors. As efficient understanding is a prerequisite for efficient review, it is compared instead. The experiment's goal is to answer two research questions:

> **RQ$_{11.1}$.** Does the use of different background colors for hunks in a two-pane diff, instead of using only a border and a common gray background, lead to more efficient understanding of code changes?

> **RQ$_{11.2}$.** Does aligning unchanged lines on the old and new side of a two-pane diff with empty space, instead of showing the old and new file without additional lines, lead to more efficient understanding of code changes?

For RQ$_{11.1}$, old code is highlighted in red and new code in green. The alignment for RQ$_{11.2}$ is reached by adding empty gray spacer blocks. Figure 11.1 depicts the four resulting combinations. Two letter codes are used in the following to denote the treatment combinations: CA for 'color' and 'aligned', CN for 'color' and 'not aligned', GA for 'gray' and 'aligned', and GN for 'gray' and 'not aligned'.

To measure understanding, the participants are asked questions regarding the code change, and the total needed time and correctness of answers is measured. Correctness is measured in three levels: Correct, partly correct and incorrect. Besides these measurements, the collected data also contains subjective assessments of the participants' opinions and values for several confounding factors (see Table 11.1).

Table 11.1: The variables collected and investigated for the diff experiment.

Independent variables (design):	
Color	dichotomous
Alignment	dichotomous
Shown code change / Task	A1/A2/A3/A4
Number of task	1/2/3/4
Independent variables (measured confounders):	
Age	ratio
Student	dichotomous
Semester (if student)	ordinal
Java experience	ratio
Current general programming practice	ordinal
Current Java programming practice	ordinal
Experience with diff viewers	ordinal
Experience with Eclipe's compare editor	ordinal
Color blindness	nominal
First or second batch	dichotomous
Dependent variables per task (measured):	
Needed time	ratio
Number of correct answers	ratio
Number of partially correct answers	ratio
Dependent variables (opinions):	
Preferred combination of color and alignment	nominal
Opinion on the chosen colors	ordinal

The experiment uses a within-groups design, i.e., every participant is tested on all four combinations of color and alignment. The same participant cannot analyze the same change twice, so that four different code changes (with associated understanding questions) are used. The participants are assigned to the four treatment groups in a round-robin fashion. The order of treatments in each group is shown in Table 11.2. The order of files was randomized for every participant.

To perform the experiment, the students created an Eclipse plugin that automated most parts of the experiment flow. Color and alignment were added to the standard compare editor of Eclipse. The time was measured manually with a stop-watch. To account for the reliance on manual measurements, the sessions were video-taped (if agreed on by the participant) to allow later assessment and correction in case of problems. Participants were asked to answer

Table 11.2: Order of treatments in the four treatment groups

Group	Treatments
A	CA → CN → GA → GN
B	GN → CA → CN → GA
C	GA → GN → CA → CN
D	CN → GA → GN → CA
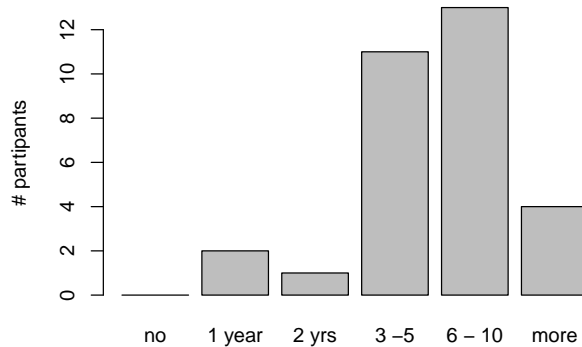
Figure 11.2: Participant's experience with Java programming (in years)

the understanding questions both quickly and correctly.

The experiment was pre-tested, and several refinements to the experiment setup were performed due to the results, for example changes to the wording of some of the understanding questions. Overall, the experimental procedure for each participant was as follows:

1. The participant enters the room and takes a seat in front of the computer.
2. A researcher explains the steps of the experiment to the participant.
3. The participant is asked for informed consent. The video recording is started if the participant agreed to be recorded.
4. The participant starts the experiment in the instrumented IDE.
5. For each of the four code changes:
   (a) The IDE shows the diff and the participant has one minute to get used to the code.
   (b) After this minute, the first question is posed to the participant.
   (c) The participant responds to the question orally and one of the researchers notes down the result.
   (d) This is repeated for a total of five questions per code change.
   (e) After answering all five questions, or after five minutes, the elapsed time is noted and the participant proceeds to the next code change. Questions that were not answered in time are counted as 'not answered'.
6. After completing the last code change, the recording is stopped and the participant answers some final demographic questions.

11.2   Results

In total, 32 participants took part in the experiment. Of these, 13 identified as students. Due to a problem with the data collection, the demographic data is incomplete for one participant and is left out in the respective statistics. One group of researchers (Koetsier, Schulz, Mergenthaler) supervised a first batch of 16 participants, another group (Fuhrmann, Leßmann) several months later the second batch of the remaining 16.

Figure 11.2 shows the Java experience of the participants. 28 of the participants have three or more years of experience with Java. 25 of the participants program more than an hour each week. Only two participants have no prior experience with a diff viewer, 20 often use one. For Eclipe's compare editor in particular, there are three participants that did not know it before and ten that use it often. One participant is red/green color-blind.
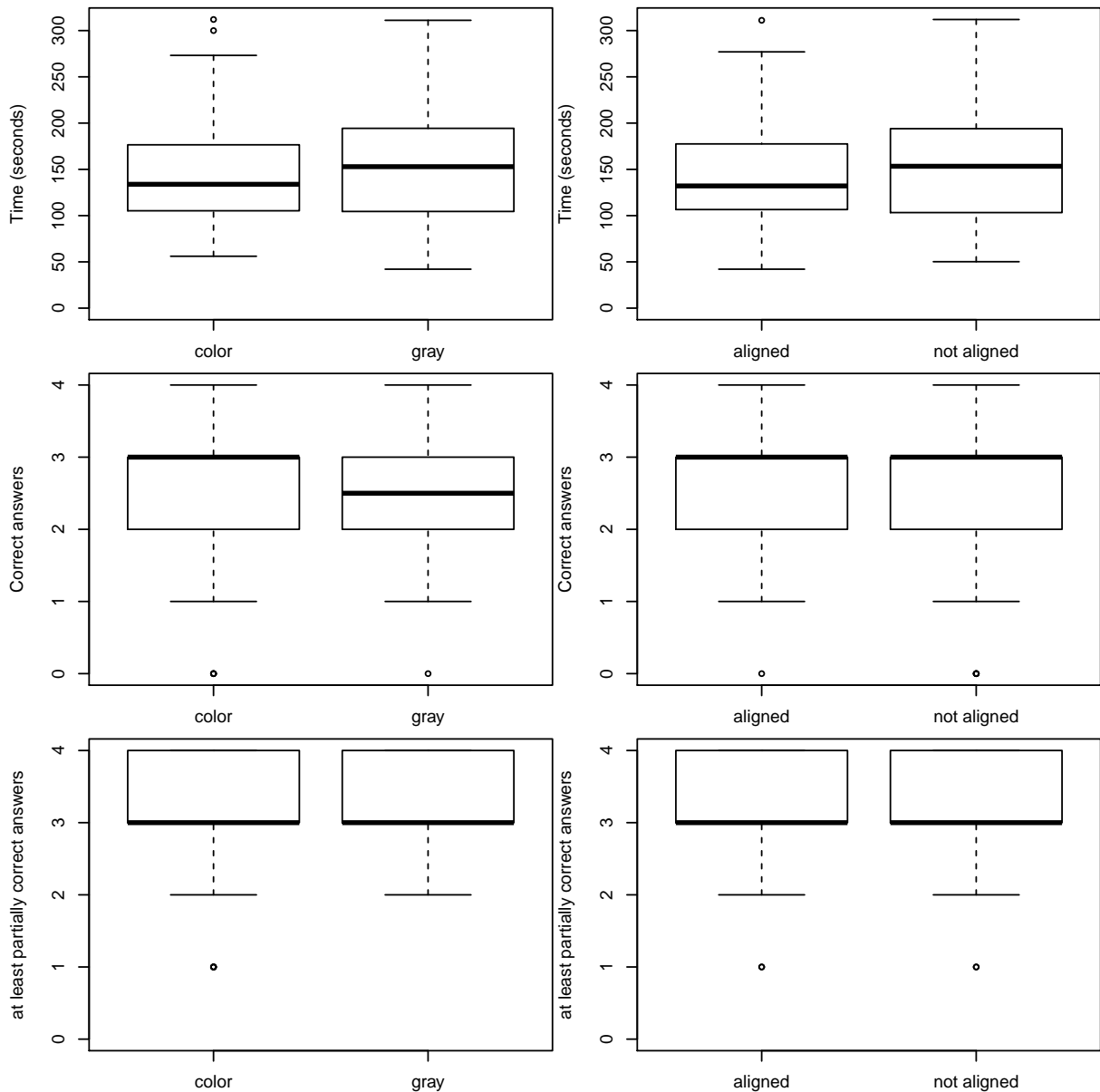
Figure 11.3: Boxplots for time and correctness of answers, dependent on color and alignment

The boxplots in Figure 11.3 provide a first overview of the differences in time and correctness, depending on the two factors of interest, color and alignment. Correctness is measured in two ways, by the number of correct answers and by the number of at least partially correct answers. The graphic indicates a small improvement in the needed time for the color condition compared to no color and for alignment compared to no alignment, but there is also a lot of variation in the data. For correctness, there does not seem to be a difference. Looking at the numerical values of the 20% trimmed means in Table 11.3 also indicates at most small effects.

Due to the complex repeated measures structure with four measurements per participant, a simple t-test is not adequate to formally check for statistical significance. A linear mixed effect model [26] is a suitable alternative. Table 11.4 shows the results from building such a model with

Table 11.3: Trimmed means, relative differences in trimmed means, and effect sizes (d) for differences in color and alignment.

	Color				Alignment			
	Gray	Color	Diff.	d[1]	Aligned	Not Al.	Diff.	d
Time	150.3 s	141.1 s	-6%	-0.17	149.1 s	142.0 s	-5%	-0.13
Fully correct answers	2.56	2.56	0%	0.00	2.60	2.52	-3%	-0.08
At least part. corr. answ.	3.20	3.42	7%	0.31	3.27	3.35	2%	0.10

[1] d := difference in trimmed means / winsorized standard deviance

Table 11.4: Results of fitting a linear mixed effect model for the needed time

Fixed Effect	Coefficient	Confidence Interval
(Intercept)	133.3 s	109.1 s ... 160.0 s
Color	-9.7 s	-26.1 s ... 7.7 s
Aligned	-8.8 s	-26.3 s ... 8.3 s
Task = A2	-11.4 s	-33.2 s ... 12.1 s
Task = A3	12.0 s	-12.5 s ... 35.1 s
Task = A4	28.7 s	4.9 s ... 53.4 s

fixed effects for color, alignment, and task, and with a random effect for the participant. It also shows the bootstrapped 95% confidence intervals for the coefficients. Both confidence intervals of interest (color and alignment) contain zero, so the effect is not statistically significant at the 5% level, even without alpha error correction. A rough estimation of the study's power, based on the observed effect size, shows that it is underpowered: More than 250 participants would be needed to achieve a power of 80% for the effect on time.

Besides these objective measures, the participants also selected their preferred treatment combination. Figure 11.4 shows the results. Here, the trend towards color and alignment is much clearer, with 22 (of 30) participants preferring the CA combination. Splitting by factors, 93% of the participants prefer color over gray-only (conf. int.: 78% – 99%) and 80% prefer alignment over non-alignment (conf. int.: 61% – 92%).

RQ$_{11.1}$: *The large majority of participants shows a subjective preference for using different colors instead of gray to highlight change parts. The objective differences are compatible with this preference, but the effect is small and it did not reach statistical significance.*

RQ$_{11.2}$: *The large majority of participants shows a subjective preference for aligning unchanged lines in the diff view by adding empty lines, instead of showing them unaligned without spacers. The objective differences are mostly compatible with this preference, but the effect is small and it did not reach statistical significance.*
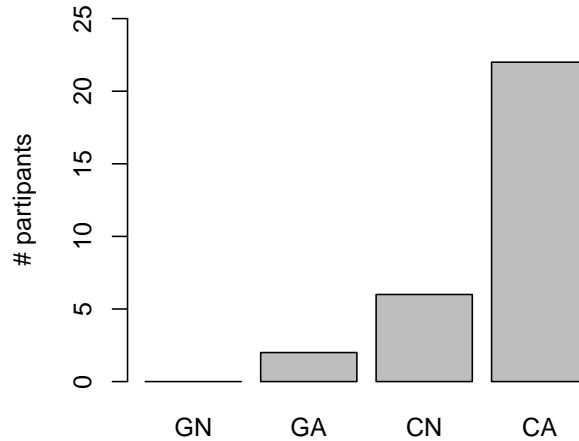
Figure 11.4: Subjective preferences for the treatment combinations

11.3 Validity and Limitations

One major threat precludes valid objective conclusions: Low statistical power. There are two options to deal with low power: Gathering data from more participants, or changing the measurement procedures to be more likely to obtain a stronger, less noisy effect. When it became clear after two batches of participants that the chosen experimental setup would need several hundred developers to obtain a significant effect, I decided to stop looking for further participants. In retrospect, there are several weaknesses in the experimental setup that reduce the measured effect: Giving the participants one minute to read the code before asking the questions swallows a part of the differences. In addition, the effort needed to understand the posed questions and to phrase an answer is a source of noise. Therefore, a replication of the experiment should carefully choose a more sensitive measurement instrument.

For the differences in the participants' subjective preferences, there are other limitations: First of all, these are just opinions not directly related to objective performance. That said, matters of taste influence tool choice and should not be neglected. In addition to this construct bias, there might also be a researcher bias, for example provoked by asking the participants for their opinion in a face-to-face setting. Prior experience with various diff viewers might have influenced the participants' opinion, too. Other factors, like color-blindness or inexperience of the participants probably introduced some noise into the data, but should not be a major problem according to the demographics presented in the previous section.

In terms of external validity, the results can probably be transferred to uses of diff viewers outside of code reviews. Intentionally, the study focused on two limited aspects of presenting diffs, so other features of diff viewers, like one-pane diffs, need to be investigated separately.

$$\diamondsuit$$

This chapter describes an experiment performed as joint work with several students to study the influence of color and alignment on the understanding of diffs in a diff viewer. No statistically significant objective results could be found, but the opinions of the participants and the tendencies in the data support two conclusions: (1) Diff viewers should use different colors to highlight the changed parts for the old and new side of the two-pane diff view. Red for old and

green for new are intuitive candidates for most participants, but it is probably best to make them configurable. (2) Diff viewers should align unchanged lines by adding spacers as needed. Several widely used diff viewers are in conflict with these recommendations: Eclipe's standard compare editor uses the exact opposite, and Atlassian Crucible and Bitbucket use color but no aligning. According to the results of the study, the diff viewer used in CoRT was augmented to use colors and alignment.

Part III

Cognitive Support: Guiding and Assisting Reviewers

12

Cognitive-Support Code Review Tools

Part I shows why improved code review support is worthwhile, and Part II describes CoRT as a platform for research on review tools. Part III of this thesis builds upon these foundations to study cognitive support for the reviewer in detail. This first chapter establishes why cognitive-support code review tools are a promising candidate for the next generation of code review tools and introduces several ways to provide such support. It combines findings from the empirical studies discussed in Part I of this thesis and from other works on code reviews and on cognitive support. The chapter is partly based on [40].

12.1 How to Improve Review Performance

This section argues that the most promising way to improve review performance is to address the reviewer's problem of understanding large changes during checking. This conclusion is reached by two independent arguments. One is bottom-up, based on empirical results on code reviews. The other is top-down, based on the theory of cognitive load.

12.1.1 Bottom-Up Argument: Factors that Impact a Reviewer's Performance

A lot of research has been done on code reviews and inspections. Still, many questions could not be answered conclusively. But some results are relatively well supported and a subset of these forms the foundation of the current chapter's discussion on how to improve review performance.

The first such research result concerns which factors have a major and which only a minor influence on the effectiveness and efficiency of reviews. When analyzing experimental data, Porter et al. *"found that [reviewers, authors, and code units] were responsible for much more variation in defect detection than was process structure"*, and they *"conclude that better defect detection techniques, not better process structures, are the key to improving inspection effectiveness."* [301] The results of Chapter 10 support this statement in a specific case. A similar conclusion is also reached by Sauer et al. [336], who identify *"individuals' task expertise as the primary driver of review performance"* based on theoretical considerations. Correlations between the (inspection) expertise of the reviewer and the number of found defects have also been reported by Rigby [317]

as well as by Biffl and Halling [54], just to name a few. I conclude that the major factors influencing code review effectiveness and efficiency are the reviewer, the reviewer's relation to the artifact under review and the way in which he or she performs the checking.

The second important result is about the role of understanding the artifact under review. In their study based on interviews with developers at Microsoft, Bacchelli and Bird found that "*[m]any interviewees eventually acknowledged that understanding is their main challenge when doing code reviews*" [18], which confirmed earlier results from Tao et al. [367]. Further support for a positive correlation between code understanding and review effectiveness comes from experiments by Dunsmore, Roper, and Wood [111]. As Rifkin and Deimel [316] put it: "*You cannot inspect what you cannot understand*". The interview results[1] fully support these findings, e.g.: "*I have to understand what the other developer thought at that time. And for that, you look very closely at the code, and then things that should or could be done better somehow come up automatically*"[I.3].

The interviewees from Part I were also asked about problems hampering review effectiveness. One of the most common themes was the difficulty in understanding and reviewing large changesets: "*Smaller commits are generally not a problem. But these monster commits are always ... not liked very much by the reviewers.*"[I.5] "*What sometimes impedes me is when the ticket is just too big.*"[I.7] "*When you have such a big pile to review the motivation is not very high and you probably don't approach the review with the needed quality in mind.*"[I.12]

The hypothesis that large, complex changes are detrimental to code review performance is reflected in many publications. An example is MacLeod et al.'s guideline to "*aim for small, incremental changes*" [241], which can be found similarly in the earlier works by Rigby et al. [318, 319]. These guidelines are based on interviews with developers and observations of real-world practices. Similar guidelines also exist for code inspection (e.g., by Gilb and Graham [140]). In the context of design inspection in the industry, Raz [315] found support for higher detection effectiveness for smaller review workloads. Rigby et al. [320] build regression models for review outcomes based on data from open-source projects. They predict review interval (i.e., time from the publication of a review request to the end of the review) and the number of found defects. The nature of their data does not permit to build models for review efficiency and effectiveness. They observe an influence of the number of reviewers and their experience, and also that an increased change size (churn) leads to an increase in review interval and number of found defects. In light of this chapter's hypotheses, the latter observation should be the result of two confounded and opposing effects: A higher total number of defects and smaller review effectiveness in larger changes.

The conclusion that large changesets are problematic can also be deduced from other research results: There is evidence that the review effectiveness greatly decreases when the review rate (checked lines of code/time for checking) is outside the optimal interval (see, e.g., [140]). Further evidence shows that concentration and therefore review effectiveness fades after some time of reviewing [218, 315]. Combining these values leads to an upper limit on the maximal size of an artifact that can be reviewed effectively in a single session.

12.1.2 Top-Down Argument: The Cognitive Load Framework

The negative effects of large changes on understanding and review performance can be derived from the construct of cognitive (over-)load from cognitive psychology, too. This section introduces the associated terms, before providing the respective argument.

---

[1]The citations are from the interviews described in Chapter 3.1 and the interviewee IDs are those from Table 3.2.

Cognitive load is "*a multidimensional construct that represents the load that performing a particular task imposes on the cognitive system*" [289]. Two parts of its substructure are the *mental load* exerted by the task and the *mental effort* spent on the task. As an example, consider the task of adding numbers in the head. The more digits the numbers have, the more mental load is exerted by the task. Now consider a young child and an older student: With the same amount of mental effort, the older student can solve summing tasks that are more complex (i.e., have a higher mental load) than those the child can solve.

Various theories use cognitive load to explain performance on cognitive tasks. An example is the cognitive load theory for learning [364], which predicts that better learning success is achieved by avoiding extraneous cognitive load.

The capacity of human *working memory* [399] greatly influences cognitive load. Cognitive psychology defines working memory as a part of human memory that is needed for short-term storage during information processing. Working memory is used when combining information, for example, when reading a sentence and determining which previously read word a pronoun refers to. The capacity of working memory in terms of distinct items is limited [82]. To overcome this limitation, items can be combined by 'chunking' [268, 348] to form new items (e.g., when consecutive words are chunked to a semantic unit). Working memory capacity can be measured using 'complex span tests' [89], in which time-limited recall and cognitive processing tasks are interleaved, and the number of correctly remembered items forms the memory span score. This score has been shown to be associated with many cognitive tasks, for example, the understanding of text [89, 90] and hypertext [101]. In the context of software engineering, Bergersen et al. [47] studied the influence of working memory on programming performance. They found that such an influence exists, yet it is mediated through programming knowledge, which, in turn, is influenced by experience. More experience allows for more efficient chunking and should, therefore, lead to lower cognitive load. In line with this prediction, Crk et al. [83] found reduced cognitive load during code understanding for more experienced participants in the analysis of electroencephalography (EEG) data.

For code review, there is evidence of the influence of expertise on effectiveness (e.g., [257]), but no studies on the influence of working memory capacity on code review performance. Hungerford et al. [177] studied cognitive processes in reviewing design diagrams and observed different strategies with varying performance. In a think-aloud protocol analysis study, Robbins et al. [322] analyzed cognitive processes in perspective-based reading. One of their observations is that combining knowledge leads to a higher number of defects found. When studying the cognitive level of think-aloud statements during reviews, McMeekin et al. [258] found that more structured techniques lead to higher cognition levels.

A different viewpoint on task performance in the presence of computerized tools is to regard human and computer as a joint cognitive system [105]. This viewpoint is also called distributed cognition. It has been proposed by Walenstein to study cognitive load in software development tools [393, 394], looking for an optimal distribution of the load among the parts of the system.

Supposing that code review is a cognitive task that follows the predictions of the cognitive load framework, code review performance depends on the reviewer's cognitive load. Overload, as well as underload, lead to sub-optimal performance. Assuming that overload is the more common situation, review performance can be improved by reducing cognitive load. The cognitive load is determined by characteristics of the subject (reviewer), the task, and by subject-task-interactions [289]. The task, in turn, consists of understanding and checking a code change, so its mental load depends on the respective sub-characteristics.
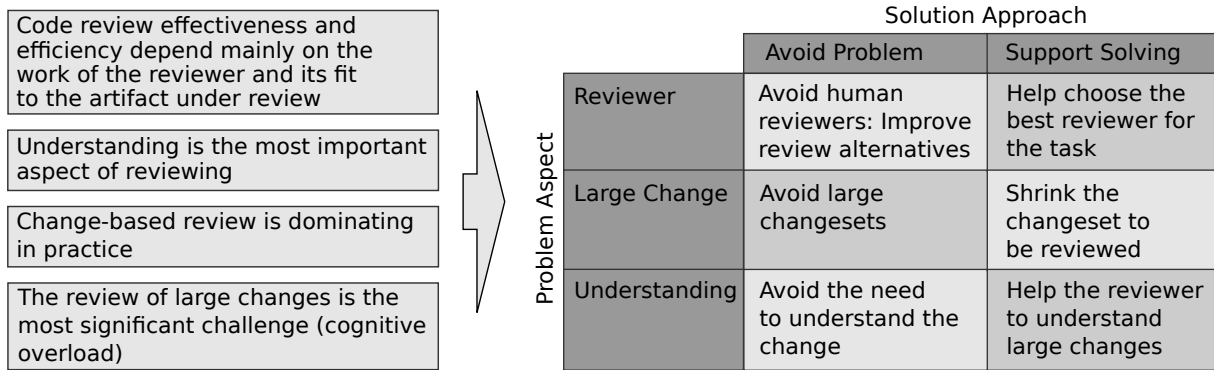
Figure 12.1: Overview of argumentation: Three main problem aspects follow from the empirical findings as well as the cognitive load theory and can be tackled in several ways.

12.2 Ideas to Address the Challenges

The previous section substantiates that to increase the effectiveness and efficiency of code reviews for defect detection, researchers should focus on the reviewer and how to help him or her to understand large code changes better. Better understanding can be achieved by reducing the cognitive load of the reviewer. Tools with features to reduce cognitive load are called 'cognitive-support code review tools' in this thesis:

> **Definition 3** (Cognitive-Support Code Review Tool). A *Cognitive-Support Code Review Tool* is a code review tool that is designed to reduce the cognitive load of the reviewer during checking.

Figure 12.1 shows an overview of the argumentation. There are three main problem aspects: The reviewer, the large change size, and the reviewer's need to understand the change. For each problem aspect, there are two general ways to deal with it: Avoid the problem aspect, or accept it and provide better support for dealing with it. The following subsections discuss each of these combinations and survey related work in these areas.

12.2.1 Problem Aspect: Reviewer

12.2.1.1 Avoid Human Reviewers

An extreme way to deal with the difficulties of human reviewers in code reviews could be to avoid human reviewers altogether. As human reviewers are one of the characteristics of code review according to Definition 1, this is synonymous to avoiding code reviews. Essentially, this is a question of efficiency: Is code review the most efficient way to find a certain defect type or are there more efficient ways, e.g., static code analysis, defect prediction, or testing? [140, 325] For example, Panichella at al. [292] found that certain problems identified during code review could have been identified earlier using static analysis. But as long as there are practically relevant defect types for which code review is most efficient, it should be used. If such defect types did not exist anymore, for example after a breakthrough in static analysis research, code review in its current form would not be needed any longer for defect detection. Therefore, this topic is not discussed further in this thesis.

12.2.1.2 Help Choose the Best Reviewer for the Task

In recent years, there have been a number of studies on 'reviewer recommendation', i.e., on finding the best reviewer(s) for a given change (e.g. [20, 376]). Although this promises a large effect, there are several problems reducing the practical benefit, especially in smaller teams. The most obvious is that in a small team, it is often fairly easy to see who is a good reviewer for a change so that computer support does not provide large gains. In some other cases, the reviewer for a certain module is fixed (see Section 6.1), so there is no choice at all. A study by Kovalenko et al. [208] supports this argument and indicates that a central use of reviewer recommendation is improved usability during reviewer selection. Additionally, always choosing the best reviewer can lead to a high review load for experienced developers, and a high workload has a negative impact on review quality [44]. Therefore, reviewer recommendation has to move from determining local optima for every single review to more global optimization of reviewer assignment. This thesis does not study reviewer recommendation further.

12.2.2 Problem Aspect: Large Change

12.2.2.1 Avoid Large Changesets

Given the problems with the review of large changes, many teams resort to the frequent review of small changes [317]. Up to a certain point, this is a good thing to do, but there are also arguments in favor of larger changes and reviews: The change under review should be self-contained, it should satisfy certain quality criteria before central check-in (at least to be compilable) and reviewing very small changes can lead to high overhead and duplicate work [367]. Hence, instead of forcing every change to be very small, the review of larger changes should be made more effective and so that changes can stay at their 'smallest natural size'.

12.2.2.2 Shrink the Changeset to be Reviewed

Given large changesets with change parts of varying relevance for the review goals, reviewers try to manually pick the relevant subset. This is regarded as hard and error-prone: *"After some time you get a feeling which files are relevant and which are not, but it's hard to filter them out. And when I don't look at them there might be some change in there that was relevant, anyway. That's problematic."*[I.8]

An important special case is systematic changes, especially rename and move refactorings. This special case has been studied for example by Thangthumachit, Hayashi and Saeki [371] as well as Ge [135]. Zhang et al. [413] describe the tool "Critics" to help inspecting systematic changes using generic templates. For the more general case, Kawrykow and Robillard [192] developed a method to identify "non-essential" differences. Tao and Kim [368] propose an approach to partition composite code changes. A lot could be gained by bringing the promising existing results into wider use. Beyond that, Chapter 15 presents research to provide a better foundation to decide which changes are low-risk. It also introduces the distinction between change parts that are error-prone and need to be checked in detail and change parts that only need to be read to help to understand other change parts. Another research path is to include more data, such as test coverage information, to assess review relevance. The latter is not examined further in this thesis and provides potential for future work.

12.2.3 Problem Aspect: Understanding

12.2.3.1 Avoid the Need to Understand the Change

From a theoretical point of view, reducing the need to understand the code is another possibility to solve the stated problem. There are defects that can be found without a deep understanding of the code, and it could be sufficient to identify these in a review if other defects are not contained or are found by other means. Chapter 15 discusses the distinction between knowledge-based and rule-based cognitive processing in this regard, but otherwise, the topic is not dealt with further in this thesis.

12.2.3.2 Help the Reviewer to Understand the Change

A theme that occurred throughout the interviews is that large changes are best reviewed with the search and hyperlinking support of an IDE (e.g., "*I think reviewing code purely in 'Crucible' only works for trivialities. Because naturally many features are missing that you have in an IDE.*"$_{I.2}$). As shown in Chapter 7, this improvement has already made its way into some widely used review tools. A benefit of IDE-integration is that new features to support understanding, e.g., symbolic execution [171], will automatically be available for reviewers, too.

Many of the interviewees try to get a high-level understanding of the change at the start of the review ("*at first an overview because otherwise, the problem is that you lose sight of the interrelation of the changes*"$_{I.10}$; this topic is taken up in Chapter 14). The support for this activity in common tools is very limited, consisting mainly of the overview of the commit messages of the singular commits belonging to the change. There is relatively little research on visualizing and summarizing code changes for better understanding: McNair, German and Weber-Jahnke propose an approach to visualize change-sets [260], as do Gomez, Ducasse and D'Hondt [145]. In addition, several textual summarization techniques have been proposed (e.g. [66]).

This thesis does not discuss summarization of changes in detail, but the thesis author (co-)supervised two master theses on this topic: Gripp [150] collected requirements for textual summarization of code changes and implemented an extension for CoRT that builds upon several of the research works mentioned above. It was successfully deployed in practice, and especially refactoring detection was considered useful by the partner company's developers. A brief glimpse at its UI is given together with CoRT's UI in Section 9.2. A second thesis by Gasparini [134] deals with graphical summaries. His study uses a mixed-methods, theory-generating design. He generated and evaluated several approaches to visualize code changes in reviews. In the empirical evaluation of prototypes for three approaches, the participants preferred an approach that made rather little use of graphical depictions. Instead, it provides a better overview of the call flow relation (cmp. Section 14.3), especially when compared to GitHub or similar simple tools. The approach was implemented and evaluated in a web-based tool based on GitHub but has not been integrated into CoRT so far.

A related technique that can help to summarize the contents of a change and to reduce its observed complexity is 'change untangling', i.e., splitting a large change that consists of several unrelated smaller changes into these smaller changes. Change untangling has first been studied by Herzig et al. [172] and alternative approaches have been proposed by Dias et al. [102], Platz et al. [299], and Matsuda et al. [253]. Barnett et al. [21] investigated change untangling in the context of code review and obtained positive results in a user study. Tao et al. [368] also proposed to use change untangling for review and showed in a user study that untangling code changes can improve code understanding.

After having an overview of the changes, the reviewer needs to step through the change parts in some order. Many reviewers try to find an order that helps their understanding, but often fall back to the order presented by their review tool: "*The problem is you sometimes get lost and don't find a good starting point.*"$_{I.10}$ "*If you don't have that, you just step through the files in the commit one after another ...*"$_{I.10}$. A similar finding resulted from a study by Dunsmore, Roper, and Wood in which participants suggested "ordering of code" to improve inspections [109]. Guiding the reviewer shares similarities with the 'step-by-step' mechanism underlying many of the reading techniques studied intensively for inspections (Section 7.1). Dunsmore et al. [107, 109, 110] developed a reading technique based on the claim that with the advent of object-oriented software development, delocalized programming plans have become more common. This also leads to *delocalized defects*, i.e., defects that can only be found (or at least found much more easily) when combining knowledge about several parts of the code. Dunsmore et al. tested their reading technique in a series of experiments. They did not find a significant influence of their technique on review effectiveness and did not analyze review efficiency. When efficiency was later analyzed in an experiment by Abdelnabi et al. [1], a positive effect was found, and the results of an experiment by Skoglund et al. [352] are largely compatible with these findings, too. The main difference between the proposed guidance and these reading techniques is that the reading techniques try to change the way the reviewer works, while the proposed guidance transfers cognitive load from the human reviewer to the tool. In addition, most reading techniques proposed so far are not intended to be used with changesets. In Chapter 14 of this thesis, it is shown in detail how ordering the code changes under review can guide the reviewer.

Clearly, the reviewer can also be supported by ensuring better comprehensibility of the code before the review, for example with code style checks [5, 28].

12.3   A New Generation of Code Review Tools

About a decade ago, Henrik Hedberg proposed a classification of software inspection/review tools into generations [167] (see Figure 12.2). He concluded that the coming fifth generation should provide flexibility with regard to the supported documents and processes and that they should comprehensively include existing research results. This prediction has come true (with limitations): Current review tools, as introduced in Section 7.3, are flexible and commonly

1: Early tools
Support of data collection

2: Distributed tools
Geographically dispersed work

3: Asynchronous tools
Meeting-less inspection

4: Web-based tools
Use of web technologies

5: Comprehensive tools
Flexibility and integration

6: Cognitive-support tools
Advanced reviewer support
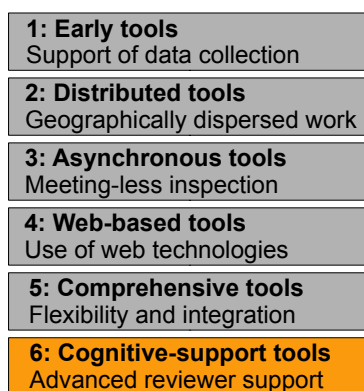
Figure 12.2: Hedberg's classification of code review tools, with cognitive support tools added as the sixth generation

support the review of any kind of text file. The preceding sections show opportunities to reach a higher level of review effectiveness. For most of them, a reification from the review of changes in text files to the review of changes in source code is necessary. The tool still allows the review of various file types, but there is much better support for some of them. I consider this to be a new generation of code review tools, the generation of *cognitive-support review tools*.

$$\diamondsuit$$

Summing up, the most promising avenue for better review tooling is to support the reviewer when understanding large changesets. This leads to the notion of 'cognitive-support code review tools', i.e., tools that are designed to reduce the cognitive load of the reviewer during checking. The next chapter further substantiates the hypothesis that reducing the cognitive load of reviewers will lead to better review performance. After that, Chapters 14 and 15 study two possibilities for cognitive support in detail: Ordering of change parts and classification of change parts.

13

An Experiment on Cognitive Load in Code Reviews

The previous chapter argued that reducing the cognitive load of the reviewer leads to better review performance. To substantiate this claim, this chapter presents a controlled experiment on aspects of cognitive and mental load during reviews. It studies the effect of working memory capacity and of code change size on review effectiveness. This chapter is based on parts of [29], joint work with Alberto Bacchelli and Kurt Schneider.

13.1 Experimental Design

The following section discusses the design of the experiment. It is also used to test hypotheses regarding the ordering of code changes. The latter parts of the experiment are detailed in the next chapter (Section 14.6).

13.1.1 Research Questions and Hypotheses

Based on the importance of human factors for improving code review performance (see Section 12.1), prior research in text and hypertext comprehension that shows an influence of working memory capacity on comprehension [101], and Chapter 12's hypotheses on a similar influence for reviews, the first research question of this chapter asks:

> **RQ**$_{13.1}$**.** Is the reviewer's working memory capacity associated with code review effectiveness?

Comparing and mentally combining different parts of the object under review may help in finding defects [177, 322] and higher working memory capacity could be beneficial in doing so. Therefore, the study looks for differences in the number of defects found. It also analyzes the subset of delocalized defects. With one-sided tests, this leads to the following null and alternative hypotheses:

$H_{1.1.0}$ There is no correlation between the total number of found defects and working memory span scores.

$H_{1.1.A}$ There is a positive correlation between the total number of found defects and working memory span scores.

$H_{1.2.0}$ There is no correlation between the total number of found delocalized defects and working memory span scores.

$H_{1.2.A}$ There is a positive correlation between the total number of found delocalized defects and working memory span scores.

Section 12.1 presented results from earlier work which brought evidence that large, complex[1] code changes are detrimental to review performance. But this evidence is mostly based on qualitative or observational data. Because a lot of research builds upon this 'large change' hypothesis, this thesis reports on more reliable support and quantitative data from a controlled setting for it:

> **RQ$_{13.2}$.** Does higher mental load in the form of larger, more complex code changes lead to lower code review effectiveness?

The hypothesis is that more complex changes pose higher cognitive demands on the reviewer, leading to lower review effectiveness. The effect on review efficiency is harder to predict: The higher cognitive load may demotivate reviewers and make them review faster (skimming), but it may also lead to longer review times. Consequently, only the effect of code change size on effectiveness is tested formally.

The probability of detection can vary greatly between different defect types. Therefore, one particular defect type was selected for this RQ: the swapping of arguments in a method call, a specific kind of delocalized defect.[2] Among the defects seeded into the code changes, three are such swapping defects: one defect in the small code change (Swap_{WU}) and two defects in one of the large code changes (Swap_{S1} and Swap_{S2}; jointly referred to as Swap_d with d as a placeholder in the following). The corresponding null and alternative one-sided hypotheses take the general form:

$H_{2.<d>.<n>.0}$ The detection probability for Swap_{WU} and Swap_d is the same when Swap_d is in the n-th review.

$H_{2.<d>.<n>.A}$ The detection probability for Swap_{WU} is larger than the detection probability for Swap_d when Swap_d is in the n-th review.

With all combinations of d='S1' or 'S2' and n='first large review' or 'second large review', there are four null and four alternative hypotheses.

13.1.2 Design

Figure 13.1 shows an overview of the phases, participation, and overall flow of the experiment. Each participant performs three reviews in total, one for a small code change and two larger

---

[1]Size and complexity are often highly correlated [164], therefore, they are not treated separately here.

[2]The experiment had to be restricted to a specific defect type to keep the experiment duration manageable and the set of defects that could be seeded into the small change was limited. I know from professional experience that swapping defects occur in practice and they can have severe consequences. I cannot quantify how prevalent they are, as studies that count defect types usually use more general categories (like "interface defects").

ones. In addition to the questions studied in this chapter, the large reviews were also used to study the effect of code ordering, which are dealt with further in Section 14.6. Next, each phase of the experiment is briefly described.

1. The experiment is entirely done through an instrumented browser-based tool that was created for the experiment to enable performing change-based reviews, collecting data from survey questions and on the interaction during reviews, and other aspects of the experiment. The welcome interface gives the participants information on the experiment and requires informed consent.

2. The participant is then shown a questionnaire to collect information on demographics and some confounding factors: The main role of the participant in software development, experience with professional software development and Java, current practice in programming as well as reviewing, and two surrogate measures for current mental freshness (i.e., hours already worked today and a personal assessment of tiredness/fitness on a Likert scale). These questions loosely correspond to the advice by Falessi et al. to measure "*real, relevant and recent experience*" [118] of participants in software engineering experiments. After providing this information, the participant receives more details on the tasks and the expectations regarding the reviews. Moreover, the participant is shown a brief overview of the relevant parts of the open-source software (OSS) project that was the source of the code changes to review.[3]

3. Each participant is then asked to perform a review on a small change; afterward the participant has to answer a few understanding questions on the code change just reviewed.

4. Next, the participant is asked to perform the first large review, preceded by a short reminder of the expected outcome (i.e., efficiently finding correctness defects). The code change in the review is ordered according to a randomly selected ordering type (see Section 14.6). Like for the small change, the participant has to answer a few understanding questions after the review.

5. Subsequently, the participant is asked to repeat the review task and understanding questions for a second large code change.

6. After all reviews are finished, the participant is asked for a subjective comparison: Which of the two large reviews was understood better? Which change was perceived as having a more complicated structure? Furthermore, the UI asks for the participant's experience with the OSS system that was the source of the code changes.

7. Finally, the participant is asked to perform an optional task of computing arithmetic operations and recalling letters shown after each arithmetic operation. This task is an automated operation span test [384], based on the shortened operation span test by Oswald et al. [288], which I re-implemented for the browser-based setting. This task is used to measure the working memory capacity for answering $RQ_{13.1}$. The task is optional for two reasons: (1) Working memory capacity as a component of general intelligence is more sensitive data than most other collected data, thus participants should be able to partially opt-out. (2) The test can be tiring, especially after having completed a series of non-trivial code reviews.

---

[3]All these descriptions could be accessed again on demand by participants during the review.
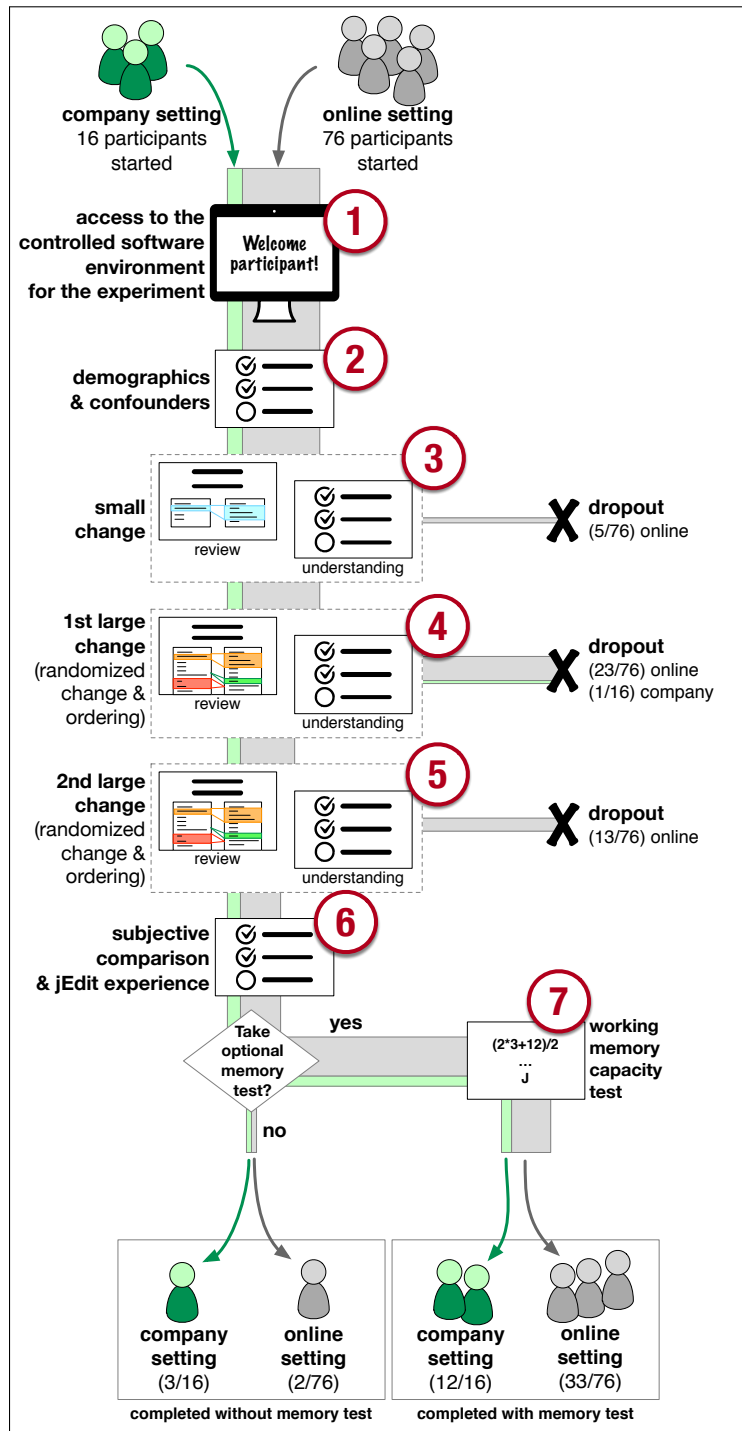
Figure 13.1: Experiment steps, flow, and participation (Source: [29])

13.1.3 Browser-Based Experiment Platform

I created a browser-based experiment environment to support all the aspects of the experiment, most importantly conducting the reviews, gathering data from the survey questions, conducting the working memory span test, and assigning participants to treatment groups. This browser-based environment allows access to professional developers and control of the ensuing threats to validity. The next paragraph details the part devised to conduct the reviews. To reduce the risk of data loss and corruption, almost no data processing was done in the UI server itself. Instead, the participants' data was written to file as log records, which were then downloaded and analyzed offline.

Part I shows that in current industrial practice, browser-based code review tools that present a code change as a series of two-pane diffs are widely used. Therefore, I implemented a similar UI for the current study. Although this setup allows for free scrolling and searching over the whole code change and thus introduces some hardly controllable factors into the design, we (Baum, Bacchelli) chose it in favor of more restrictive setups because of its higher ecological validity. During code review, the UI logged various kinds of user interaction in the background, for example, mouse clicks and pressed keys.

An example of the review UI can be seen in Figure 13.2. The HTML page for a review starts with a description of the commit. After that, a brief description of the UI's main features was given, followed by the change parts in the order chosen for the particular review and participant. The presentation of each change part consisted of a header with the file path and method name and the two-pane diff view of the change part. The initial view for a change part showed at most four lines of context below and above the changed lines, but the user could expand this context to show the whole method. Further parts of the code base were not available to the participants (and not needed to notice the defects). Review remarks could be added and changed by clicking on the margin beneath the code.

13.1.4 Objects and Measurement

Patches / Code changes. Because it is infeasible to find a code base that is equally well known to a large number of professional developers, the study uses one that is probably little known to all participants. This increases the difficulty of performing the reviews. To keep the task manageable, at least the functional domain and requirements should be well known to the participants and there should be little reliance on special technologies or libraries. To satisfy these goals, the jEdit programmer's text editor[4] was selected as the basis from which to pick code changes to review. To select suitable code changes, I screened the commits to jEdit from the years 2010 to 2017. I programmatically selected changes with a file count similar to the median size of commercial code reviews identified in previous work [21, 41]. The resulting subset was then checked manually for changes that are (1) self-contained, (2) neither too complicated nor too trivial, and (3) of a minimum quality, especially not containing dubious changes/"hacks".

With that procedure, I manually checked ten commits for the large reviews and excluded two as not self-contained, one as too complicated, two as too simple and three as dubious. The selected commits are those of revision 19705 ('code change A' in the following) and of revision 19386 ('code change B'). In addition, I selected a smaller commit (revision 23970) for the warm-up task. For this choice, a large number of commits satisfied the criteria, so the sampling was less exhaustive. Code change A is a refactoring, changing the implementation of various file

---

[4]http://jedit.sourceforge.net

Commit description

Allow columns to be rearranged in file system browser

Code changes

Below you find the code changes to review. The old version of the code is on the left, the new version is on the right.

To add a review remark, click on the respective line number. To delete it, click on it again and delete the remark's text. If a defect spans multiple lines, just mark one of those lines. If similar defects appear multiple times, please mark every occurrence. If you suspect something could be a defect but are not 100% sure, it's better to add a review remark.

At several of the change parts, you can show the whole changed method by clicking on "(Show more context)".

org/experiment/editor/browser/VFSDirectoryEntryTable.java, constructor

```
66          setDefaultRenderer(Entry.class,
67              renderer = new FileCellRenderer());
68
69          header = getTableHeader();
70          header.setReorderingAllowed(false);
71      addMouseListener(new MainMouseHandler());
72      header.addMouseListener(new MouseHandler());
73      header.setDefaultRenderer(new HeaderRenderer(
74
    (DefaultTableCellRenderer)header.getDefaultRenderer()));
```

org/experiment/editor/browser/VFSDirectoryEntryTable.java, constructor
(Show more context)

```
66          setDefaultRenderer(Entry.class,
67              renderer = new FileCellRenderer());
68
69          header = getTableHeader();
70          header.setReorderingAllowed(true);
71      addMouseListener(new MainMouseHandler());
72      header.addMouseListener(new MouseHandler());
73      header.setDefaultRenderer(new HeaderRenderer(
74
    (DefaultTableCellRenderer)header.getDefaultRenderer()));
```

org/experiment/editor/browser/VFSDirectoryEntryTableModel.java

```
275
```

org/experiment/editor/browser/VFSDirectoryEntryTableModel.java, columnMoved()

```
275      protected void columnMoved(int from, int to) {
276          if (from == to)
277              return;
278          if (from < 1 || from >= getColumnCount())
279              return;
280          if (to < 1 || to >= getColumnCount())
281              return;
282          ExtendedAttribute ea = extAttrs.remove(from - 1);
283          extAttrs.add(to - 1, ea);
284      }
285
```

org/experiment/editor/browser/VFSDirectoryEntryTable.java

```
574      class ColumnHandler implements TableColumnModelListener
575      {
576          public void columnAdded(TableColumnModelEvent e) {}
577          public void columnRemoved(TableColumnModelEvent e) {}
578          public void columnMoved(TableColumnModelEvent e) {}
579          public void columnSelectionChanged(ListSelectionEvent e)
{}
580
581          public void columnMarginChanged(ChangeEvent e)
582          {
```

org/experiment/editor/browser/VFSDirectoryEntryTable.java
(Show more context)

```
574      class ColumnHandler implements TableColumnModelListener
575      {
576          public void columnAdded(TableColumnModelEvent e) {}
577          public void columnRemoved(TableColumnModelEvent e) {}
578          public void columnMoved(TableColumnModelEvent e) {
579
    ((VFSDirectoryEntryTableModel)getModel()).columnMoved(
                    e.getToIndex(), e.getFromIndex());
580
581          }
582          public void columnSelectionChanged(ListSelectionEvent e)
{}
583
584          public void columnMarginChanged(ChangeEvent e)
585          {
```

End review ▶

Figure 13.2: Example of the review view in the browser-based experiment UI, showing the small code change. It contains three defects: In 'VFSDirectoryEntryTableModel', the indices in the check of both 'from' and 'to' are inconsistent (local defects). Furthermore, the order of arguments in the call in 'VFSDirectoryEntryTable.ColumnHandler' does not match the order in the method's definition (delocalized defect). At each defect, there is a review remark marker (little red square) in the line number margin, i.e., the figure could show the view at the end of a review that found all defects. (Source: [29])

Table 13.1: Code change sizes, complexity, and number of correctness defects (total defect count as well as count of delocalized defects only) after seeding

Code Change	Changed Files	Change Parts	Presented LOC[1]	Cycl. Compl.[2]	Total Defects	Delocalized Defects
Small / Warm-up	2	3	31	12	3	1
Large code change A	7	15	490	57	9	2
Large code change B	7	21	233	83	10	3

[1] presented LOC := Lines Of Code visible to the participant on the right (=new) side of the two-pane diffs without expanding the visible context

[2] total cyclomatic complexity [255] of the methods on the right (=new) side of the two-pane diffs

system tasks from an old API for background tasks to a new one. Code change B is a feature enhancement, allowing the combination of the search for whole words and the search by regular expressions in the editor's search feature. The small change is also a feature enhancement, allowing columns to be rearranged in the editor's file system browser UI. Details on the sizes of the code changes can be found in Table 13.1. Mainly because it contained a lot of code moves, code change A contains fewer change parts but more lines of code than code change B, but code change B is algorithmically more complex.

There was a risk that some participants would ignore the given instructions and look for further information about jEdit (e.g., bug reports) on the internet while performing the experiment. To reduce that risk all mentions of jEdit and its contributors were removed from the code changes presented to the participants and credit to jEdit was only given after the end of all reviews. I also normalized some irrelevant, systematic parts of the changes (automatically added @Override annotations, white space) and added some line breaks to avoid horizontal scrolling during the reviews, but otherwise left the original changes unchanged.

Seeding of Defects. Code review is usually employed by software development teams to reach a combination of different goals (see Chapter 4). Of these goals, the detection of faults (correctness defects; [178]), improvement of code quality (finding of maintainability issues), and spreading of knowledge are often among the most important ones. As the definition of maintainability is fuzzy and less consensual than that of a correctness defect, the analysis of the experiment is restricted to correctness defects. In its original form, code change A contained one correctness defect and code change B contained two. To gain richer data for analysis, several further defects were added, so that the small code change contains a total of 3 defects, code change A contains 9 defects, and code change B contains 10. The seeded defects are a mixture of various realistic defect types. There are rather simple ones (e.g., misspelled messages and forgetting a boundary check), but also hard ones (e.g., a potential stall of multithreaded code and a forgotten adjustment of a variable in a complex algorithm). Six of these defects are delocalized [107], i.e., their detection is likely based on combining knowledge of different parts of the code. Two researchers independently classified defects as (de)localized and contrasted results afterward. Both types of defects can be seen in Figure 13.2: The parameter swap in the call of 'columnMoved' is a delocalized defect because both the second and third change part have to be combined to find it. The off-by-one errors in the if conditions in the second change part are not delocalized because they can be spotted by only looking at that change part. The full code changes and the seeded defects can be found in the study's replication package [42].

Measurement of Defects. The experiment UI explicitly asked the participants to review

only for correctness defects. All review remarks were manually coded, taking into account the position of the remark as well as its text. A remark could be classified as (1) identifying a certain defect, (2) 'not a correctness defect', or (3) occasionally also as relating to several defects. In edge cases, a remark was counted if it was in the right position and could make a hypothetical author aware of his defect. It was not counted if it was in the right position but unrelated to the defect (e.g., it is related only to a minor issue of style). If this procedure led to several remarks for the same defect for a participant, it was only counted once. To check the reliability of the coding, a second researcher (Ghofrani) coded a sample of 56 of the remarks again. In 13 of these cases, it was discussed whether the remark should be counted as a defect or not. In all cases, it was agreed that the original coding was correct. The detailed coding of all review remarks can be seen in the study's online material [42].

Working Memory. As described in Section 12.1.2, working memory capacity can be measured with complex span tests [89] that consist of interleaved time-limited recall and cognitive processing tasks. The implementation of the shortened automated operation span test [288, 384] consists of two tasks each with 3 to 7 random letters. Each letter is shown for a brief amount of time, and the letter sequence has to be remembered while solving simple arithmetic tasks after each letter (see the experiment's online material for more details [42]). Each correctly remembered letter gives one point, so the maximum score is 50. The theoretical minimum is zero, but this is unlikely for the study's population. Rescaling the results from Oswald et al. [288] gives an expected mean score of 38.2. Before the main tasks, there were some tasks for calibration and getting used to the test's UI.

13.1.5 Statistical Data Analysis

For $RQ_{13.1}$, correlation between working memory span scores and found defect counts is checked using Kendall's τ_B correlation coefficient [4]. τ_B is used because it does not require normality and can cope with ties, which are likely for counts. In line with the hypotheses for this RQ, tests are performed for the total number of found defects as well as the total number of found delocalized defects. To augment and triangulate the results, I also build a regression model to predict the number of found (delocalized) defects. The independent variables are chosen by stepwise selection (stepwise BIC [388]) from the variables listed in Table 13.2.

For $RQ_{13.2}$, the experiment returned simple count data (defect found in the small review, defect found in the large review) that leads to 2x2 contingency tables. As the observations are dependent (several per participant), McNemar's exact test [3] is used.

13.1.6 Participants

With a large number of participants it is infeasible to use a code base that is well known to all participants. Also, inexperienced participants (e.g., students) would be overwhelmed by the difficult review tasks. The choice of an online, browser-based setting helped to increase the chance of reaching a high number of professional software developers. Because this meant less control over the experimental setting, the experiment contains questions to characterize the participants:

- Their role in software development,
- their experience with professional software development,
- their experience with Java,
- how often they practice code reviews, and

Table 13.2: The variables collected and investigated for the cognitive load + ordering experiment.

Independent variables (design):	
Working memory span score (measured)	ordinal/interval
Used change part order type (controlled, see Section 14.6)	nominal
Used code change (controlled)	dichotomous
First or second large review (controlled)	dichotomous
Independent variables (measured confounders):	
Professional development experience	ordinal/interval
Java experience	ordinal/interval
Current programming practice	ordinal/interval
Current code review practice	ordinal/interval
Working hours before experiment (surrogate for tiredness)	ratio
Perceived fitness before experiment	ordinal
Experience with jEdit	ordinal
Screen height	ratio
Controlled setting (i.e., lab instead of online)	dichotomous
Dependent variables per review:	
Needed gross review time	ratio
Needed net review time (i.e., without pauses)	ratio
Number of detected defects	ratio
Number of detected delocalized defects	ratio
Number of correctly answered understanding questions	ratio

- how often they program.

The experiment UI was made available online in 2017 for six weeks. Similar to *canary releasing* [176], I initially invited a small number of people and kept a keen eye on potential problems, before gradually moving out to larger groups of people. To contact participants, we (Baum, Bacchelli) used our personal and professional networks and spread the invitation widely over mailing lists, Twitter, Xing, Meetup and advertised on Bing. The complete development team of the partner company participated in the experiment. This subsample of 16 developers performed the experiment in a more controlled setting, one at a time in a quiet room with standardized hardware. This subpopulation allowed us to detect variations between online setting and a more controlled environment that can hint at problems with the former.

A consequence of the chosen sampling method is that participants could not be assigned to groups in bulk before the experiment, but had to be assigned with an online algorithm. The assignment was performed with a combination of balancing over treatment groups, minimization based on the number of defects found in the small change review (Point 3 in Figure 13.1) and randomization.

As the mean duration turned out to be about one and a half hours, with some participants taking more than two hours, we (Baum, Bacchelli, Schneider) decided to offer financial compensation. This was done in form of a lottery [213, 349], offering three cash prizes of EUR 200 each. The winners were selected by chance among the participants with a better than median total review efficiency so that there was a mild incentive to strive for good review results.

A total of 50 participants finished all three reviews (of 92 who submitted at least the warm-up review). 45 chose to also take the working memory span test. Unless otherwise noted,
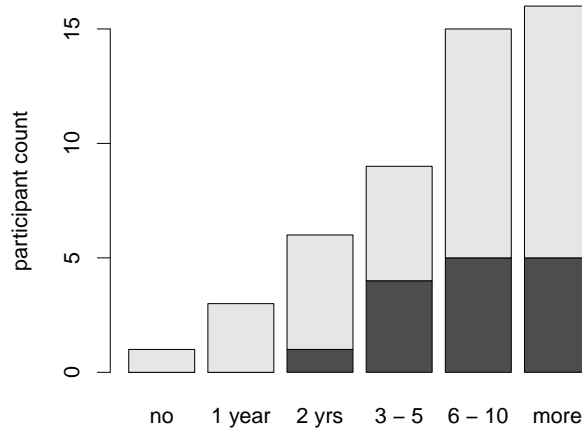
Figure 13.3: Professional development experience of the 50 participants that finished all reviews. Darker shade indicates company setting, lighter shade is pure online setting.

Table 13.3: Mean and standard deviation (sd) for the number of defects found (all Defects as well as the subset of delocalized defects only) and review time, depending on review number and code change

	All Defects	Delocalized Defects	Time (Minutes)
Small / Warm-up	1.76 (of 3), sd=1.13	0.76 (of 1), sd=0.43	8.49, sd=5.48
First large review	3.68, sd=2.12	0.66, sd=0.82	30.03, sd=16.41
Second large review	2.98, sd=2.16	0.68, sd=0.87	20.69, sd=17.32
Large code change A	3.16 (of 9), sd=2.09	0.7 (of 2), sd=0.86	23.85, sd=16.96
Large code change B	3.5 (of 10), sd=2.23	0.64 (of 3), sd=0.83	26.88, sd=17.93

participants who did not complete all reviews are not taken into account in further analyses. Participants who spent less than 5 minutes on a review and entered no review remark are regarded as 'did not finish'. Another participant was excluded because he or she restarted the experiment after having finished it partly in a previous session. 24 participants dropped out during the first large review, 13 during the second large review.

42 participants named 'software developer' as their main role. Four are researchers and the remaining participants identified as managers, software architects or analysts. 47 of the participants program and 32 review at least weekly. 40 have at least three years of experience in professional software development, only one has none. Figure 13.3 shows the detailed distribution of experience. None of the participants ever contributed to jEdit.

The minimum observed working memory span score is 17, the maximum is 50, the median is 45 and the mean is 41.93 (sd=7.05). The mean in the study's sample is about 3.7 points higher than estimated based on the sample of Oswald et al. [288] (from a different population), and there seems to be a slight ceiling effect that is discussed in the threats to validity (Section 13.3).

13.2 Results

This section presents the empirical results and the performed statistical tests. Before that, it briefly describes general results on the participant's performance and their qualitative remarks. The complete dataset is available [42].

Table 13.4: Mean review efficiency and effectiveness for the two levels of control in the study (online or company setting)

Controlled Setting	Effectiveness	Efficiency
Yes (Company)	43%	8.8 defects/hour
No (Online)	36%	10.4 defects/hour

Table 13.3 shows the mean number of defects found and the mean review time depending on the reviewed code change and also depending on the review number (i.e., small review, first large review or second large review). This preliminary analysis indicates a large ordering/fatigue effect, particularly striking for the drop in review time between the first and second large review. Comparing the company and the online setting, the mean review effectiveness in the company is higher than online, whereas the efficiency is slightly smaller (see Table 13.4).

The participant's full-text answers were analyzed for potential problems with the experiment. Most comments in this regard revolved around some of the compromises in the design of the experiment: Little review support in the web browser (e.g., *"For more complex code as in review 2 or three a development IDE would support the reviewer better than a website can do"* $_{PC11}$[5]), mentally demanding and time-consuming tasks (e.g., *"I found the last exercises complicated."* $_{PO33}$), and the high number of seeded defects (e.g., *"I am rarely exposed to code that bad"* $_{PO44}$). There were also a number of positive comments (e.g., *"It was quite fun, thanks!"* $_{PO37}$).

I also scanned the participants' scrolling and UI interaction patterns. These patterns indicate that some participants made intensive use of the browser's search mechanism, whereas others scrolled through the code more linearly. The detailed patterns are available with the rest of the experiment results [42].

13.2.1 Working Memory and Found Defects

$RQ_{13.1}$ asks: "Is the reviewer's working memory capacity associated with code review effectiveness?" As motivated in Section 13.1.1, all defects as well as delocalized defects are analyzed. Kendall's τ_B rank correlation coefficient (one-sided) is used to check whether a positive correlation exists. For all defects, τ_B is 0.05 (n=45, p=0.3143). Looking only at the correlation between the number of delocalized defects found and working memory span, τ_B is 0.24 (p=0.0186); inversely it is almost zero (-0.01) for localized defects. Using an alpha value of 5% and applying the Bonferroni-Holm procedure for alpha error correction, the null hypothesis $H_{1.2.0}$ can be rejected for delocalized defects, yet with a rather small Kendall correlation. The null hypothesis $H_{1.1.0}$ for all defects cannot be rejected.

Looking at the scatter plot in Figure 13.4 helps to clarify the nature of the relation: The plot suggests that a high working memory span score is a necessary but not a sufficient condition for high detection effectiveness for delocalized defects. In other words, a higher working memory capacity seems to increase the chances of finding delocalized defects, but it does not guarantee it and other mediating or moderating factors must be present. It may seem that the leftmost data point is a very influential outlier and its exclusion would indeed reduce statistical significance; however, systematic analysis of influential data points showed that it is not the most influential

---

[5]The subscripts next to the citations are participant IDs, with POn from the online setting and PCn from the more controlled company setting.
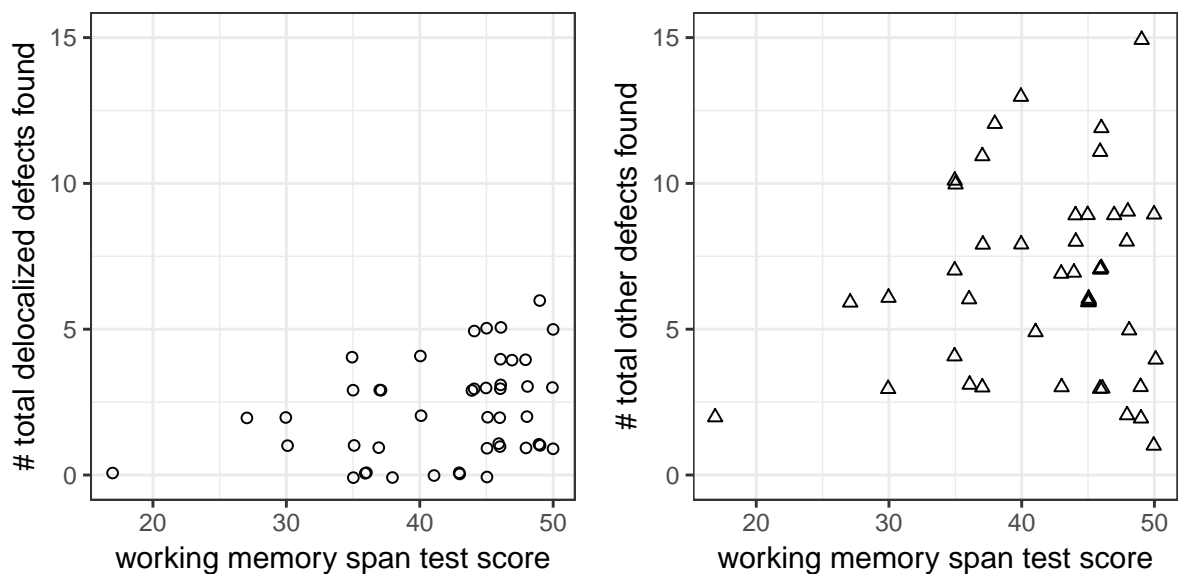
Figure 13.4: Scatter Plots of Working Memory Span and Number of Delocalized (Left Plot) and Other (i.e., Localized; Right Plot) Defects Detected; Slight jitter added

Table 13.5: Results of fitting a linear regression model for the number of all found defects using stepwise BIC. Coefficient for time is based on measuring in minutes.

Coefficients		Model statistics		
Intercept	Net time for reviews	R^2	adj. R^2	BIC
4.7803	0.0682	0.2768	0.2599	258.61

one. The three most influential participants both for and against the hypothesis were scrutinized, checking the time taken, found defects, answers to understanding questions and other measures. Based on these in-depth checks, we (Baum, Bacchelli) decided to stick to the formal exclusion criteria described in Section 13.1.6 and keep all included data points. If we were to exclude one influential data point, it would be participant PO15, who spent little time on the reviews; this exclusion would strengthen the statistical significance.

Other influencing factors were checked for with a regression model fitted by stepwise BIC (Bayesian Information Criterion) [388]. Additionally, the correlation between all factors and the defect counts (see Table 13.7) was determined. The only effect strong enough for inclusion in the regression model for all defects was *review time*, with longer reviewing time leading to more defects found (see Table 13.5). Looking at the correlations for the remaining factors in Table 13.7, professional development experience and review practice also seem to be positively correlated with the total number of defects found, as is the experimental setting. Review practice and experimental setting can also be found in the regression model for delocalized defects (Table 13.6). The other factors, e.g., subjective mental fitness/tiredness, are neither highly correlated nor did they make it into one of the regression models.

Table 13.6: Results of fitting a linear regression model for the number of all found delocalized defects using stepwise BIC. Coefficient for time is based on measuring in minutes.

Coefficients					Model statistics		
Intercept	Net time for reviews	Working memory span score	Current review practice	Controlled setting	R^2	adj. R^2	BIC
-3.6914	0.0215	0.0865	0.2717	0.8478	0.5359	0.4895	162.54

Table 13.7: Kendall τ_B correlation for all variable combinations. The names are replaced by single letter IDs for space reasons: a := Working memory span score, b := Total number of detected defects, c := Total number of detected delocalized defects, d := Total number of detected localized (=other) defects, e := Net time for reviews, f := Controlled setting, g := Professional development experience, h := Current code review practice, i := Screen height, j := Current programming practice, k := Java experience, l := Code change B in first large review, m := Working hours before experiment, n := Perceived fitness before experiment

	b	c	d	e	f	g	h	i	j	k	l	m	n
a	0.05	0.24	−0.01	0.07	−0.18	0.04	−0.01	0.13	−0.19	0.22	−0.04	−0.2	0.23
b		0.64	0.88	0.35	0.28	0.24	0.2	0.21	0.07	0.04	−0.05	−0.08	−0.08
c			0.45	0.44	0.3	0.24	0.31	0.22	0.1	0.07	−0.01	−0.07	0.11
d				0.28	0.2	0.21	0.13	0.2	0.03	0.02	−0.06	−0.08	−0.12
e					0.21	0.15	0.13	0.09	−0.07	0.01	−0.12	−0.06	−0.09
f						0.02	0.28	0.3	0.13	−0.07	0.01	0.25	0.02
g							0.14	0.04	0.24	0.5	0.17	−0.05	−0.05
h								0.05	0.47	−0.02	0.03	0.07	−0.04
i									0	0.05	−0.06	−0.05	0.08
j										0.09	0.07	−0.04	−0.19
k											0.05	−0.15	−0.05
l												−0.09	0.07
m													−0.13

> **RQ**$_{13.1}$: *Working memory capacity is positively correlated with the effectiveness of finding delocalized defects. Not delocalized defects are influenced to a much lesser degree, if at all. Even for delocalized defects, working memory capacity is only one of several factors, of which the strongest is review time.*

13.2.2 Code Change Size and Found Defects

RQ$_{13.2}$ asks: "Does higher mental load in the form of larger, more complex code changes lead to lower code review effectiveness?" It could already be seen in Table 13.3 that the reviewers performed better in the small review than in the larger reviews. More specifically, the mean review effectiveness is 59% for the small reviews and 35% for the large reviews. The mean review efficiency is 15.65 defects/hour for the small review and 9.47 defects/hour for the large

Table 13.8: Count of reviews in which the respective defects were detected, p-value from one-sided McNemar's test for $RQ_{13.2}$ and corresponding effect size measured as Cohen's g [77] (classification as 'large' also according to Cohen [77])

Hypothesis	Defect found in review				p	Cohen's g
	small only	large only	both	none		
$H_{2.S1.first\ large\ review}.0$	10	0	9	6	0.001	0.5 (large)
$H_{2.S2.first\ large\ review}.0$	11	0	8	6	0.0005	0.5 (large)
$H_{2.S1.second\ large\ review}.0$	10	1	9	5	0.0059	0.41 (large)
$H_{2.S2.second\ large\ review}.0$	12	1	7	5	0.0017	0.42 (large)

reviews. These numbers depend to a large degree on the seeded defects. For a fair comparison, a specific defect type was picked to compare in detail (as described in Section 13.1.1): The swapping of arguments in a method call. The small code change contained one defect of this type and code change A contained two such defects (S1 and S2 in the following). The defect has been found in the small reviews in 38 of 50 occasions (detection probability: 76%). When code change A was the first large review, S1 has been found 9 of 25 times (detection probability: 36%). The detailed numbers for all four situations can be seen in Table 13.8. The corresponding tests for association are all highly statistically significant (the smallest threshold, after applying Bonferroni-Holm correction, is 0.05/4=0.0125; the largest p-value is 0.0059). Therefore, all four null hypotheses $H_{2.<d>.<n>}.0$ can be rejected, with the conclusion that the probability of finding 'swap type' defects is smaller for larger, more complex code changes. In all four situations, the effect size is large. There is only a small difference in effect size between the first and second large review, i.e., if there is a fatigue or other ordering effect it does not play a major role.

Part of the hypothesis underlying $RQ_{13.2}$ is that the lower performance is due to increased mental load. This increase should have a larger effect for the participants with lower working memory span score, so one would expect a higher drop in detection effectiveness for them. I checked for such an effect, but it is far from statistically significant. All in all, it is not possible to reliably conclude whether the lower effectiveness is due to cognitive overload in spite of high mental effort or to a decision to invest less mental effort than needed (e.g., caused by lower motivation).

$RQ_{13.2}$: *Larger, more complex code changes are associated with lower code review effectiveness. This may be caused by higher mental load or by other reasons, such as faster review rates or lower motivation.*

13.3 Validity and Limitations

This section discusses the threats to validity of the results from the current chapter.
External Validity. A setup similar to code review tools in industrial practice, code changes from a real-world software system, and mainly professional software developers as participants strengthen the external validity of the experiment. There is a risk of the participants not being as motivated as they are in real code reviews, which was tried to counter by making code

review efficiency part of the precondition to winning the cash prize. External validity is mainly hampered by four compromises: (1) Usually, code reviews are performed for a known code base. (2) Unlike in industry, discussion with other reviewers or the author was not possible in the reviews. (3) The defect density in the industry is usually lower than in the experiment's code changes. (4) The experiment UI asked participants to focus on correctness defects, although identification of maintainability defects is normally an important aspect of code review [248, 374]. This could pose a threat to external validity if the mechanisms for finding other types of defects are notably different. This threat is probably under control, as there are delocalized design/maintenance issues with a need for deeper code understanding as well, but this claim has to be checked in future research. It would also be worthwhile to study whether and to what extent the high defect density often used in code review experiments is a problem.

Construct Validity. To avoid problems with the experimental materials, a multi-stage process was used: After tests among the authors, I performed three pre-tests with one external participant each. Afterward, the canary release phase started.

Many of the used constructs are defined in previous publications and the experiment reuses existing instruments as much as possible, e.g., the automated operation span test and many of the questions to assess the participants' experience and practice. It was not formally checked whether the used implementation of the operation span test measures the same construct as other implementations, but this threat is mitigated since far more diverse tests have been shown to measure essentially the same construct [399]. Compared to working memory, the mental load construct is less well defined and usually assessed using subjective rankings.

One of the central measures in the current study is the number of defects found, which was restricted to correctness defects to avoid problems with fuzzy construct definitions. To reduce the risk of overlooking defects that a participant has spotted, the experiment UI asked participants to favor mentioning an issue when they are not fully sure if it really is a defect. This advice is compatible with good practices in industrial review settings [140]. The defects were seeded by the author of this thesis, based on his (then) 11-years experience in professional software development and checked for realism by another researcher (Bacchelli). Still, there can be implicit bias in seeding the defects as well as in selecting the code changes. Also, a defect is either considered delocalized or not, which ignores that the distance between the delocalized parts differs depending on the defect and change part order. This could increase noise in the data.

A sample of 50 professional software developers is large in comparison to many experiments in software engineering [351]. For other sources of variation, the experiment had to be limited to considerably smaller samples, leading to a risk of mono-operation bias and limited generalizability: For example, there are only three different code changes, and only one analyzed defect type for $RQ_{13.2}$. Similarly, the set of analyzed defects and defects types for $RQ_{13.1}$ is limited to a small sample of all possible defects and defect types, and there might well be other defect types whose detection is influenced by working memory capacity. In addition, there could be alternative commonalities between the defects besides 'delocalization' that also explain the found correlation.

Internal Validity. A threat to validity in an online setting is the missing control over participants, which is amplified by their full anonymity. To mitigate this threat, the experiment included questions to characterize the sample (e.g., experience, role, screen size). To identify and exclude duplicate participation, the experiment UI logged hashes of participant's local and remote IP addresses and set cookies in the browser. This threat was further controlled by

comparing the online and company sub-sample. There are signs that the participants in the controlled setting showed less fatigue and more motivation, i.e., the difference in review time between first and second large review is less pronounced than in the online setting. The experiment UI asked the participants to review in full-screen mode and did not mention jEdit, but it is still possible that participants in the online setting searched for parts of the code on the internet. If somebody did, this would increase the noise in the data.

The participant sample is self-selected. Many potential reasons for participation make it more likely that the sample contains better and more motivated reviewers than the population of all software developers. Probably, this does not pose a major risk to the validity of the main findings; on the contrary, I would expect stronger effects with a more representative sample. The participant's working memory span scores were higher than those observed in other studies [288, 384]. The resulting slight ceiling effect might have reduced statistical power in the analyses that use working memory span scores. This effect could be due to the selection bias[6] and possibly also to a general difference in working memory span scores between software developers and the general population. Still, it could also be a sign of a flaw in the implementation of the working memory span test. A downside of having the working memory span test at the end is that the study cannot detect a measurement error caused by only measuring participants that are tired due to the reviews. Given the above-average working memory test results of the participants compared to other studies, this does not seem to be a major problem.

Statistical Conclusion Validity. Ideally, the experiment should show a causal relationship between working memory capacity and review effectiveness for $RQ_{13.1}$. This demands controlled changes to working memory capacity [295], which is ethically infeasible. Therefore, it was checked for potential confounders (see Table 13.2) but unobserved confounders cannot reliably be ruled out and only associations are reported.

$$\diamondsuit$$

Summing up, this chapter shows that working memory capacity is positively correlated with the effectiveness of finding delocalized defects, and that larger, more complex code changes are associated with lower code review effectiveness. Both findings support the argument that lower cognitive load leads to better review performance. This provides a foundation for the studies of detailed support mechanisms in the next two chapters.

---

[6]This explanation is supported by the negative correlation between company/online setting and working memory (Table 13.7)

14

Ordering of Change Parts

As motivated in Chapter 12, cognitive-support review tools should help the reviewer to understand a code change better and faster. The current chapter argues that one way to do so is to determine an order of reading the code that helps the reviewer understand it. The tool can then use this order to guide the reviewer. First, it is explored whether the order of code changes used by the tool is indeed used as a guidance by the reviewers. Then, a theory-generating mixed-methods study results in principles for how such an optimal order could look like. An important input for determining an optimal order are the 'relations' between the change parts. The found principles are specified as a formal theory and implemented in software. Selected predictions of the theory are tested in a controlled experiment, an extension to the experiment of the previous chapter. Furthermore, the approach is implemented in CoRT and used in the partner company. The theory-generating study has been published in [41] and the experiment in [29], both joint works with Alberto Bacchelli and Kurt Schneider.

14.1 Methodology

This section describes the overall methodology for this chapter in more detail. The detailed information for the individual studies is contained in the respective sections.

14.1.1 Research Questions

Although an iterative methodology has been used, the results are structured linearly along four research questions.

Reviewers can navigate the code in two ways: On their own, driven by hypotheses they form along the way, or guided by the order presented by the tool. There is qualitative evidence that both occur in practice and that the current tool order is sub-optimal (Section 12.2.3.2). To add further qualitative as well as quantitative support to this claim, $RQ_{14.1}$ (Section 14.2) asks:

> $RQ_{14.1}$. How relevant to reviewers is the order of code changes offered by the code review tool?
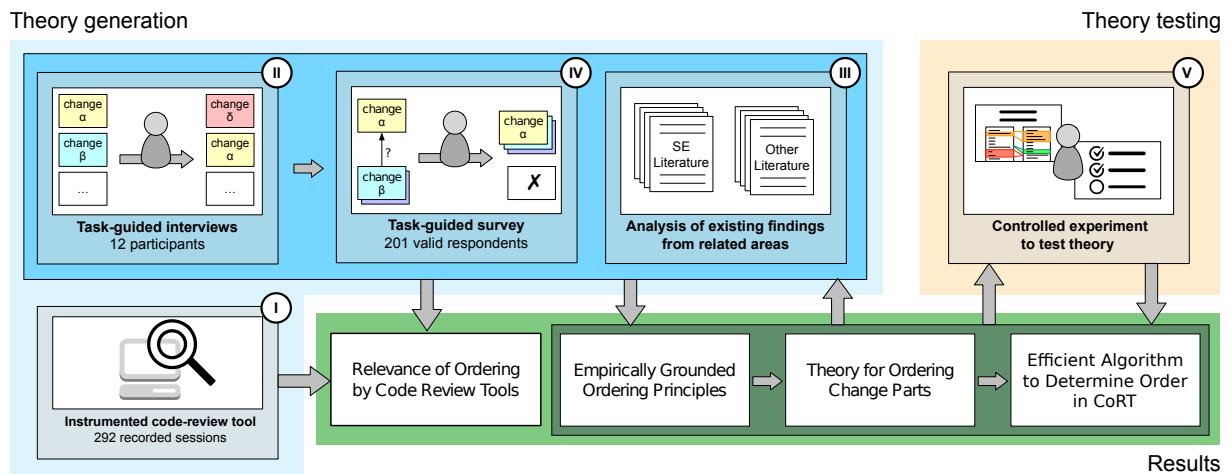
Figure 14.1: High-level view of the research method (parts based on [41])

Given that the current order is perceived as sub-optimal, the next question then focuses on empirically defining what makes a better order (Section 14.3):

> **RQ**$_{14.2}$**.** Are there principles for an optimal order of reading code changes under review?

Those principles, if any, would provide a human-readable impression of what is meant when talking about presenting the changes in a better order. To be implemented in software or used for predictions, they need to be specified more precisely (Section 14.5):

> **RQ**$_{14.3}$**.** How can the notion of 'better order for code review' be formalized as a theory?

Appendix D shows how the problem of finding an optimal tour according to the theory from RQ$_{14.3}$ can be solved in polynomial time. The claims of the theory can be tested formally in a controlled experiment (Section 14.6):

> **RQ**$_{14.4}$**.** Can the order of presenting code change parts to the reviewer influence code review efficiency, and does this depend on working memory capacity?

14.1.2 Research Method

This chapter's theory-generating approach was inspired by methods to iteratively generate theory from data, most notably Grounded Theory [143, 144]. The study started with interviews as a flexible means to gather rich data and triangulated the findings with empirical results from related fields as well as with an online survey. Furthermore, I collected log data from code reviews in the partner company with CoRT to support the claim that better automatic ordering of changes can help to improve review. After formalizing the theory, it was tested in a controlled experiment and implemented in CoRT. Figure 14.1 details the data sources and their connection to the results.

Table 14.1: Interview participants: ID, experience, and changeset

ID	Dev. exp. (in years)	Change set	ID	Dev. exp. (in years)	Change set
OI1	2	A	OI7	23	B+C
OI2	8	A	OI8	3	A
OI3	3	A	OI9	10	A
OI4	3	A	OI10	5	A
OI5	7	A	OI11	10	A
OI6	5	B+C	OI12	10	A

Logged review navigation (Point I in Figure 14.1): There is qualitative evidence from the interviews in Part I and also from others [21] that the current standard order of review tools, i.e., alphabetical by path, is not optimal. To triangulate these findings and to gain more quantitative information, I analyzed the navigation patterns from the interaction data that was logged by CoRT in the partner company (see Section 9.3). The data used in the current study consists of 292 reviews collected during fall 2016. To determine the size of code changes, I analyzed the company's versioning system. The data set is available [43].

Task-guided Interviews (Point II): The main method of data collection in the early phases of the study was a special form of *task-guided* interview. I prepared exemplary code changes from real world projects and printed each part of the changes in form of a two-pane diff on a different piece of paper. Interviewees were asked to sort the shuffled parts for a change into the order believed to be best for review. A short printed description of the participant's task and the purpose of the change was also handed out. While sorting, the participant was asked to think aloud. When the participant had finished, the interviewer explicitly asked for a rationale for the given order. After that, the interviewer presented an alternative order (either taken from an earlier interview or prepared by the researchers before the session) and the participant was asked to explain why his/her order was better than this alternative. In most interviews, this comparison was repeated with yet another order, in other interviews the whole procedure was repeated with a different code change.

The study used three different changesets (A, B, and C) sampled from real-world software systems. The main criteria for selecting the changes were their size/complexity (manageable, yet not trivial) and their content (based on the preliminary hypotheses). Changes consist of 7 to 10 change parts, mainly in Java files, but also in XML and XML Schema files. In the interviews, the participants were familiar with the code base for changes B and C, but not for A. Details on the changes are in the study's material [43].

All of the sampled interview participants had good programming knowledge, but not all of them were experienced reviewers. Table 14.1 presents the participants' experience. Seven of the interviews were performed by the author of this thesis and five by a co-author of the study (Bacchelli). In the first four interviews, the researcher took interview notes; all of the other interviews were recorded and later transcribed.

The interviews resulted in two types of data: the orders declared optimal by the participants and the interview transcripts. The interview transcripts were analyzed by open coding augmented by memoing. The codes were then refined and checked in a peer card sort session [158, 357] performed jointly by both interviewers, followed by further selective coding. The sequences given by the participants were included in the card sorting and also analyzed programmatically

to systematically search for nonrandom patterns. Furthermore, they were used later to check the formalization of the theory. During the whole research process, I used memoing to capture ideas and preliminary hypotheses.

Existing Findings in Related Areas (Point III): One of the guidelines in Grounded Theory is that "*other works simply become part of the data and memos to be further compared to the emerging theory*" [143]. With this mindset, existing findings from related areas were used to inform and triangulate the theory. The selection of related fields was guided by theoretical sampling, e.g., by looking for works on hypertext after the importance of relations began to emerge.

Task-based Survey (Point IV): After formulating preliminary hypotheses, we (Baum, Bacchelli) conducted an online survey. It contained *task-guided* confirmatory questions to challenge the preliminary hypotheses and exploratory questions to develop the theory further. We defined the target population as 'software developers with experience in change-based code review' and included a set of questions to filter respondents accordingly.

We used established guidelines for survey research [183] to formulate the questions and structuring the survey. The main part consisted of tasks asking respondents to declare their opinion regarding different code orders for review. In addition, besides the filter questions, the survey contained three questions on the participant's navigation behavior during code review (for $RQ_{14.1}$) and four questions to analyze potential confounding factors, such as the used review tool and programming language. All the questions were optional, except for the filter questions.

The main, task-guided part consisted of four pages; Figure 14.2 shows an extract of one. Every page started with the abstract description of a code change (Point 1 in Figure 14.2) and ended with a free text question for further remarks (Point 2). Between that, a selection of three types of questions was included, which asked for: the participant's preferred order of the change parts (Point 3), a comparison of two orders pre-selected based on the research hypotheses (Point 4), and an assessment of the usefulness of a set of orders on a 4-point Likert scale (not shown in Figure 14.2). The order in which the change parts and proposed orders were presented was randomized.

We used eight pre-tests with software developers to iteratively optimize the survey. The creation and testing of the survey took seven weeks; the final survey ran for five weeks. To invite software developers to the survey, we randomly sampled active GitHub users and invited developers from our professional networks. We invited a total of 3,020 developers. The initial filter questions were answered by 238 people (response rate: 8%), of which 201 were part of the target population. Not all participants completed the survey or answered all questions. We excluded participants if values for the respective hypothesis/research question were missing or if an answer was inconsistent with one of their earlier answers. Therefore, the total number of answers differs for the analyses.

Of the respondents, 97% program and 85% review at least weekly, so differences in practice of the participants are of little importance. 57% of the respondents have only one or two years of experience with regular code reviews; these participants were included unless otherwise noted, but only after statistically showing their answers to be distributed like those of the respondents with 3 years or more of experience. The influence of sampling through GitHub is clearly visible: A majority (102 of 177) uses GitHub pull requests for reviewing and JavaScript (66 of 133) or Java (41 of 133) as a programming language. 89% (117 of 132) develop software commercially and 72% (95 of 132) participate in open source. The survey and the data sets are available [43].

Controlled Experiment (Point V): In a controlled experiment, the theory was tested by

Figure 14.2: Example main page from the survey (data-flow variant/Situation 2a) (Source: [41], © IEEE)

comparing the review performance in a specific situation depending on the order of the code. Details on the design of the experiment are given in Section 14.6, after having introduced the generated theory.

14.2   The Relevance of the Order by the Tool

$RQ_{14.1}$ seeks to understand the relevance of the ordering of changes proposed by the code review tools. To answer this question, the study builds on opinions gathered in interviews, log data from CoRT, and estimates from the survey's respondents.

The log data contains traces of files visited in a review session. By comparing these files to the alphabetical order, they show whether the user followed the order of the review tool, which was alphabetical at that time. In 156 of 292 studied review sessions (53%), the user started with the file presented first by the review tool. When reviewing further, 3,071 of 8,254 between-file navigations (37%) took the reviewer to the next file in the tool order; moreover, in 162 (55%) of the review sessions, the reviewer visited additional files that were not part of the changeset. The numbers suggest that the hyperlinking and search features of the IDE might help the reviewer navigating, after a place to start is found.

For small changes, the presentation order may have a negligible effect, because the number

How often do you **start** a review with the change presented **first** by the tool?

| 6 | 4 | 25 | 54 | 43 |

count

How often do you **continue** with the change presented **next** by the tool?

| 7 | 9 | 29 | 63 | 24 |

count

Never Rarely Sometimes Often Always
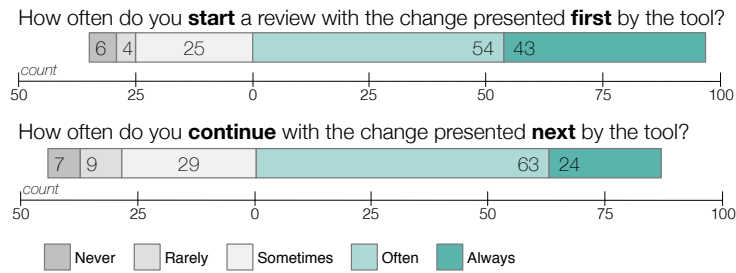
Figure 14.3: Survey results: Relevance of the ordering offered by the tool (Source: [41], © IEEE)

of permutations and the cognitive load for the reviewer grows with change size. In a study at Microsoft, Barnett et al. [21] found a median size of 5 files and 24 diff regions in changes submitted for review. The review sessions at the partner company are even larger, with a median size of 11.5 files per task. About 54% of the reviews have a scope of 10 or more files. There is no statistically significant pattern connecting review size and navigation behavior.

The interviewers asked for the participants' opinion on an alphabetical order for review. The participants were either neutral or negative about it, e.g., *"Well, I don't think file based is a good order [...] or alphabetical order is definitely not a good order."*$_{OI9}$ *"I mean it's clear that GitHub doesn't have any intelligence behind the way that it presents you the reviews, currently, so even a small improvement is welcome."*$_{OI10}$ And although the survey did not ask for it, some participants left a similar remark, for example: *"I've never thought about ordering of changes in code review tools. But while filling this survey I started thinking that proper ordering could make reviewing of code much simpler."*$_{OS270}$

When a code review is performed jointly with the author of the code, the author can guide the reviewer through the code. Therefore, the survey asked about joint reviews. Of 167 participants answering this question, 32 (19%) perform reviews 'often or always' together with the code's author. The survey asked the remaining participants about their behavior in two situations: (1) When starting the review and (2) when in the midst of the review. 132 respondents answered these questions and a large fraction reported to use the tool's order 'often or always': 97 (73%) for the start and 87 (66%) for the middle of the review; Figure 14.3 reports the details. There is a tendency for more experienced reviewers and for reviewers with IDE-integrated review tools to use the tool's order less often, but the general picture stays the same for all studied sub-populations.

Summing up, in a significant number of change-based code reviews, the reviewers use the order in which the code changes are presented by the review tool to step through the code, although they report to regard this order as sub-optimal for efficient understanding and checking of the code. The size of a code change under review is often small, but in a notable number of cases not small enough to make the effect of the order irrelevant. The problem is more pronounced for less knowledgeable reviewers. A code review tool should, therefore, present the changes in an order that is well-suited for a human reviewer. The results of the next research questions describe how such an order should look like.

14.3 Principles for an Optimal Ordering

14.3.1 General Principles

The interview participants believe that certain orders are better suited for code review than others. But the choice of the optimum is subjective. However, participants generally acknowledged that other orders are good, too, and believe that a number of orders will be similar in terms of review effectiveness and efficiency: "*I don't necessarily think this is worse. It's more a different point of view.*"$_{OI4}$ "[This order] *probably makes sense if you're super-deep into the system.*"$_{OI10}$

Following the 'tour/path' metaphor used in other publications [285, 337], the term 'tour' is used in the following to denote a permutation of the change parts under review. 'Change parts' and change hunks from the version control system are related, but do not have to be identical, as is further detailed in Section 14.5.

Based on the combination of the different data sources, it emerges that an order that obeys the principles described in the following is perceived to lead to better review effectiveness and efficiency compared to other orders.

> **Principle 1.** Group related change parts as closely as possible.

By grouping related change parts together, a good tour reduces context switches and reduces the cognitive load for the reviewer. Additionally, it eases the task of comparing neighboring change parts to spot inconsistencies or duplications: "*So here, seems a bit like a code clone.* [...] *And this is actually why I think it is really cool to have these two* [related change parts] *together.*"$_{OI10}$ "*If I was to return on this one, I would have to switch the context, which is bad.*"$_{OI8}$ "*I think a review tool should try to group changes that 'logically' belong together.*"$_{OS44}$ "*Unrelated things should not get in the way of related things.*"$_{OS296}$ As shown in Table 14.2, Principle 1 is also well supported in the survey results.

> **Principle 2.** Provide information before it is needed.

This allows the reviewer to better understand the change parts: "*Without the knowledge if this attribute is required or optional, I can't tell if the mapper is correctly implemented.*"$_{OI7}$ "*Blue depends on green, so it's useful to know what green is before reviewing blue.*"$_{OS248}$

Table 14.2: Survey results: Confirmatory questions for Principle 1.

Preference	Sit. 2a + 2b[1]	Sit. 2a (Only data-flow)	Sit. 2b (Only attr. decl.-use)	Sit. 4[2]
close	102	67	35	102
not close	14	4	10	6
no preference	7	4	3	
Total responses	123	75	48	108

[1] Only one of Situation 2a (data-flow relation) and Situation 2b (declaration-use relation) was shown selected by chance.

[2] Results for Situation 4 are deduced from the user-given order (Point 3 in Figure 14.2), therefore "no preference" does not occur.

Table 14.3: Survey results: Comparison of different ways to order change parts related by call-flow (one callee, four callers), used for Principles 2 and 6.

Order strategy	Count among best rated	Count among worst rated	Mode (prevalent answer)
bottom-up (i.e., callee first)	111	16	very useful (100 times)
top-down breadth-first	35	61	somewhat useful (49 times)
top-down depth-first	34	67	somewhat useful (56 times)
no sensible rule	10	112	not very useful (55 times)[1]
Total responses	130	130	130

[1] For experienced reviewers, the mode is "not at all useful" (21 of 45).

Table 14.4: Survey results: Confirmatory questions for Principle 3.

Preference	Sit. 2a + 2b[1]	Sit. 2a (Only data-flow)	Sit. 2b (Only attr. decl.-use)
prefer closeness	82	57	25
prefer direction	26	12	14
no preference	11	5	6
Total responses	119	74	45

[1] Only one of Situation 2a (data-flow relation) and Situation 2b (declaration-use relation) was shown selected by chance.

Although the survey did not include a confirmatory question for Principle 2, one of the results might be attributable to it: The respondents showed a clear tendency towards going bottom-up along the call-flow relation, with 111 of 130 (85%) rating bottom-up as preferred (see Table 14.3).

> **Principle 3.** In case of conflicts between Principles 1 and 2, prefer Principle 1 (grouping).

When reviewers come across a change part where they need knowledge they do not yet have, they need to make assumptions (at least implicitly). As long as the related information-providing change parts are coming shortly afterward, they can then check the assumptions against reality: *"The order doesn't actually influence me that much."*$_{OI11}$ *"Maybe the order was not what I preferred, but the groupings of the snippets made sense."*$_{OI5}$ *"The only thing that matters here is that purple and gold appear one after the other, whichever first."*$_{OS392}$ Principle 3 is supported in the survey, although less than Principle 1: 76% of the respondents who indicated a preference preferred closeness (see Table 14.4).

> **Principle 4.** Closely related change parts form chunks treated as elementary for further grouping and ordering.

Principle 4 is needed to explain some of the interview results and is supported in the literature on cognitive processes, but it did not emerge explicitly from the interviewees' statements. Therefore, the survey included two exploratory questions to investigate it (Situation 3). The

tour using chunking to let the relations point in the preferred direction was chosen as better by 35 of 50 respondents (70%) in one and 38 (76%) in the other question.

> **Principle 5.** The closest distance between two change parts is *"visible on the screen at the same time."*

Seeing two closely related change parts directly after another is good, but seeing them both at the same time is better: With the latter, the cognitive load is minimal and inconsistencies can be spotted. As the participants put it: *"The most useful presentation would be to display all* [5 related] *changes at once and to allow the user to navigate freely"*$_{OS44}$ *"I'd prefer to have them both* [...] *on screen, ideally."*$_{OS54}$ *"I'd make* [the description] *stay on top, wherever I look at."*$_{OI8}$

> **Principle 6.** To satisfy the other principles, use rules that the reviewer can understand. Support this by making the grouping explicit to the reviewer.

An order that the reviewer does not understand can *"break his line of thought"* and lead to disorientation. Making the grouping explicit helps the reviewers to understand it, to form expectations and to divide it into parts they can handle separately. *"We're going back from something that is more specific to something that is generic. And that kind of breaks my line of thought."*$_{OI10}$ *"*[This order] *doesn't have a specific pattern, at least none that I can immediately identify.* [...] *This is bad"*$_{OI5}$ *"If the parts had been grouped, the groups made visible and ideally given sensible names, I would have been able to understand the ordering better."*$_{OI5}$

When asking the survey participants to rate bottom-up vs top-down tours, there was also a tour that was not based on a "sensible" rule. This option was rated among the worst by 112 of 130 respondents (86%), thus supporting Principle 6.

The common guideline to 'keep commits self-contained' is a special case of Principle 1 combined with Principle 6. In this case, a commit is an explicit group of related change parts.

The importance of a change part for code review varies, e.g., some change parts are more defect-prone than others. The participants took this importance into account to varying degrees. Some used it as the prime ordering criterion, while others did not. A lot of the variation in participant's orders from the interviews is due to these differences in the assessment and handling of unimportance. The importance of change parts for review is dealt with in Chapter 15 of this thesis.

14.3.2 The Macro Structure: How to Start and How to End

At the very beginning of the review, the reviewer should learn about the requirements that led to the change. Many also wanted to get some kind of overview at the start (*"First introduction to understand the context, then the crucial part"*$_{OI2}$). An example of usage, e.g., a test case, can help to achieve this. Such overviews are further studied in the master theses of Gripp [150] and Gasparini [134].

The interviewees use several tactics (T) how to proceed after the initial overview, i.e., start with: (T1) something easy, (T2) a natural entry point, e.g., GUI, Servlet or CLI, (T3) the most important change parts, (T4) new things, (T5) change parts that "don't fit in", if any. Of these tactics, T1 and T2 often suit Principle 2 better, i.e., to provide information before it is needed. In contrast, T3, T4 and T5 are heuristics to visit more important/defect-prone change parts early.

Some participants gave tactics for the end of the review, too: (1) End with a wrap-up/overview (e.g., a test case or some other example of usage putting it all together), or (2) put the unimportant rest at the end.

14.3.3 The Micro Structure: Relations between Change Parts

Principle 1 states that related change parts should be close together. The participants gave a number of different types of 'relatedness', e.g.: (1) Data flow, (2) call flow, (3) class hierarchy, (4) declare & use, (5) file order, (6) similarity, (7) logical dependencies, and (8) development flow. A more detailed description of the relation types can be found in the study's supplemental material [43].

Most of these relations are inherently directed (e.g., class hierarchy or data flow), while others are undirected (e.g., similarity). For many of the directed relations, there is a preferred direction (e.g., to put the declaration of an attribute before its use); for others—mainly for call flow—the preferred direction seems to be more subjective. Many interview participants prefer to go top-down from caller to callee, but others also talked about going bottom-up from callee to caller. In contrast, the survey results support bottom-up (see Table 14.3). This indicates that a simple global rule of "always prefer bottom-up/top-down" probably does not exist.

Another distinction between the relations is whether they are binary or gradual. For a binary relation, like call flow, there are only two possibilities: Either there is a relation or there is none. For a gradual relation, like similarity, the distinction between related and unrelated is fuzzier.

Fregnan's master thesis [129] built upon the identified relations and asked a sample of 18 developers to rank the relation types by importance. The 'method call' relation type (a subtype of 'call flow') was considered most important by the developers.

14.4 Input from Other Research Areas

This section discusses related work on the ordering of change parts for review and compares it to the found principles.

14.4.1 Reading Comprehension for Natural Language Texts

Brain regions responsible for language processing are also active during code comprehension [347]. There are differences in the activation patterns between code and text, but these become less pronounced with programming experience [124]. Therefore, I used several studies from the large body of research on reading comprehension and the understandability of natural language texts to inform the ordering theory.

The "Karlsruhe comprehensibility concept" [146], an extension of the "Hamburg comprehensibility concept" [220], summarizes multiple studies on factors influencing the comprehensibility of natural language texts. It names six influencing factors/dimensions: Structure, concision, simplicity, motivation, correctness and perceptibility. This chapter's approach to improve code ordering mainly targets the 'structure' dimension.

The positive impact of a sensible, explicitly recognizable or presented text structure is also reported in other studies (e.g., [128, 148, 267]). A good text structure is "coherent", i.e., parts of the texts stick together in a meaningful and organized manner [148]. Reading scrambled paragraphs takes more time, and is detrimental to recall quality when there is a time limit [200].

Presenting news and corresponding explanations in a clustered way can improve the understanding and interest of a reader [409]. And there is evidence that stories are mentally organized in a hierarchical fashion [57].

The aforementioned results fit to those from the previous section, but there is also some evidence to the contrary: McNamara et al. found that a less coherent structure can improve the learning of knowledgeable readers from a text, presumably because they have to think more actively [261]. The same could be true in the case of reviews, with a sub-optimal structure forcing the reviewer into a more active role. This underlines the need to empirically test the predictions in a controlled experiment.

14.4.2 Hypertext: Comprehension and Tours

The ordering theory is based on the assumption that the relations between change parts are an important factor in determining the optimal tour. The resulting part graph shares many similarities to a hypertext. Hypertext research has studied how different link structures and different presentations of the structure influence a reader's interest and understanding (e.g., [408]). In the case of reviews, the link structure cannot be influenced, but the presentation can, and a hierarchical presentation has been found to be beneficial [303]. It has also been found that characteristics of the reader, notably working memory capacity and cognitive style, mediate the influence of structure [101].

The notion of 'guided tours' has also been proposed for hypertext [379]. In addition, Hammond and Allison [157] suggest the metaphors of "go-it-alone" (similar to the targeted navigation briefly mentioned in Section 14.1.1) and of "map navigation" (similar to the participants' need to get an overview). An approach to automatically create such guided tours has been proposed by Guinan and Smeaton [152]. They, too, use patterns to determine the order of the nodes.

14.4.3 Empirical Findings on Real-World Code Structure

Developers in long-living projects likely try to structure their code in a way that helps understanding. Therefore, I looked for empirical results on the order of methods and fields in software systems. I found two: Biegel et al. observed that in many cases, the code adheres to the structure specified in the Java Code Conventions published by Sun/Oracle, and that a clustering by visibility is also quite common [51]. They could also observe semantic clustering (by common key terms), whereas alphabetic order was rare. Geffen and Maoz studied a number of different criteria, also on open-source Java projects but with a stronger focus on call-flow relationships [137]. They found that a "calling" criterion of having a callee after the caller (i.e., top-down) is often satisfied. Regarding the conflicting results on top-down vs bottom-up between the interviews and survey, this can be regarded as a point in favor of top-down. The article of Geffen and Maoz also contains results on a second study: They tested experimentally whether clustering or sorting by call-flow helps to understand code faster. Their results are not statistically significant, but they show a tendency that a random order is worst and a combination of clustering and sorting is best, especially for inexperienced developers.

14.4.4 Clustering of Program Fragments and Change Parts

After the importance of grouping in an optimal tour became clear, I started to look at existing approaches for clustering in software. Many clustering approaches exploit structural [247] and similarity relations [210], possibly augmented with latent semantic analysis [245]. Often

clustering is performed using randomized meta-heuristics, an approach that is incompatible with Principle 6 (understandable rules). In contrast, the ACDC approach of Tzerpos and Holt [383] is based on recognizing patterns in subsystem structures and encouraged me to pursue a similar approach. Most clustering approaches deal with modules. In contrast, 'change untangling' (see Section 12.2) can be seen as a special kind of clustering of change parts.

14.4.5 Program Comprehension: Empirical Findings and Theories

A number of theories on the cognitive processes of developers during code comprehension have been proposed. Developers sometimes use 'bottom-up comprehension', i.e., they combine and integrate parts of the program into increasingly complete mental models. On other occasions, they employ 'top-down comprehension', either inference-based by using beacons in the code or expectation-based guided by hypotheses [284]. They switch between these modes depending on their knowledge, the needs of the task, and other factors [387, 389]. The survey by Storey [362] provides further information.

Recent studies looked at the navigation behavior of developers during debugging and maintenance. It was found that 'information foraging theory' provides a more accurate pattern of navigation behavior than hypothesis-driven exploration [229, 230]. In information foraging, developers follow links between program fragments. These links are largely based on dependencies. The importance of links/relations for developer navigation has also been noted in other studies [132, 203, 367]. A comparison of developers with differing experience showed that effective developers navigate by following structural information [323] and make more use of chunking [227].

Storey et al. [361] combined empirical findings from the literature to derive guidelines for tools that support program comprehension. The theory described in this chapter results in such a tool, therefore their guidelines should be partly reflected in it. Specifically, two of Storey et al.'s guidelines for bottom-up comprehension are to "reduce the effect of delocalized plans" and "provide abstraction mechanisms". They are addressed by grouping and by hierarchical chunking, respectively. By allowing the reviewers to also explore the source code on their own, further of Storey et al.'s guidelines can be satisfied.

14.5 A Theory for Ordering the Change Parts to Review

The principles and findings described in the previous sections detail what makes a good order of changes for code review, but to implement them in software or to test the hypotheses, they are still too vague. Therefore, they are formalized based on the guidelines for building theories in software engineering by Sjøberg et al. [350], i.e., by giving the theory's scope, constructs, propositions, and the underlying explanations.

14.5.1 Scope and Constructs

The scope of the theory is regular, change-based code review. The theory has been developed based on code written in object-oriented languages; it possibly has to be adapted to be applicable to other programming paradigms. The constructs of the theory are detailed in Table 14.5.

Table 14.5: Definitions of the constructs for the ordering theory

Construct	Description
Code change	The 'code change' consists of all changes to source files performed in the 'unit of work' (see Definition 2 on page 24) under review. This also includes auxiliary sources like test code, configuration files, etc. The code change defines the scope of the review, i.e., the parts of the code base that shall be reviewed. With task or user story level reviews, a code change can consist of multiple 'commits'.
Review efficiency	Review efficiency is the number of defects found per review hour invested (definition adapted from [52]).
Review effectiveness	Review effectiveness is the ratio of defects found to all defects in the code change (definition adapted from [52]).
Defect	In the context of this theory, a defect is any kind of true positive issue that can be remarked in a review. This encompasses faults as defined in the IEEE Systems and Software Engineering Vocabulary [178], but also for example maintenance issues.
Change part	The elements of a code change are called 'change parts'. In its simplest form, a change part corresponds directly to a change hunk as given by the Unix diff tool or the version control system. When some part of the source code was changed several times in a code change, a change part can span more than two versions of a file. It could be beneficial to split large change hunks into several change parts, e.g., when the hunk spans several methods.
Tour	A tour is a sequence (permutation) of all change parts of a code change.
Relation (between change parts)	There can be 'relations' between change parts. A relation consists of a type (e.g., call flow, inheritance, similarity; see Section 14.3.3) and an ID that allows distinguishing several relations of the same type (e.g., the name of the called method). There are relations of differing strength, but these are not taken into account in the current formal model. Change parts (as vertices) and relations (as edges) define a graph with labeled edges, the 'part graph'. There are directed as well as undirected relations. Undirected relations can be modeled as two directed edges so that the graph is directed. There can be multiple edges between two change parts, but their labels have to be distinct. The approach further demands that the graph has no loops. A mechanism similar to the one used by Barnett et al. [21] can be used to get from the syntactic level to the relation graph.
Grouping pattern	The grouping and ordering preferences of a reviewer are modeled as 'grouping patterns'. A grouping pattern combines a matching rule that identifies a subset with at least two change parts in the part graph and a function *rate* to provide a rating for a permutation of the matched change parts. Only one family of grouping patterns was sufficient to describe the data so far: A 'star pattern' (see Figure 14.4) matches a core vertex and all vertices (at least one) that are connected by an edge with a given relation type and the same ID to the core. In the 'bottom-up' case, the rating function assigns a high rating (e.g., 1) to all sequences that start with the core and a low rating (e.g., 0) to all others.
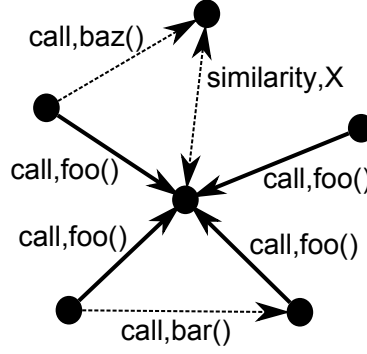
Figure 14.4: Example of a star pattern (thick edges) in a change part graph (Source: [41], © IEEE)

14.5.2 Propositions

The goal of this section is to define a partial order $\geq_T \subseteq Tour \times Tour$ (in words: is better than or equal) between tours that captures the notion of the utility of an order for review. For every pair of tours t_1 and t_2 it holds (other things being equal):

$$\forall t_1, t_2 : t_1 \geq_T t_2 \Rightarrow (\text{reviewEfficiency}(t_1) \geq \text{reviewEfficiency}(t_2) \wedge$$
$$\text{reviewEffectiveness}(t_1) \geq \text{reviewEffectiveness}(t_2))$$

The proposition above states that a tour that is better than another in terms of $\geq_T$ will not be worse in terms of review efficiency or effectiveness. A stronger proposition shall also hold, namely that there are tours where a better ranking in terms of $\geq_T$ means better review efficiency (with $>_T$ defined as usual: $x >_T y \Leftrightarrow (x \geq_T y \wedge \neg (y \geq_T x))$):

$$\exists t_1, t_2 : t_1 >_T t_2 \Rightarrow \text{reviewEfficiency}(t_1) > \text{reviewEfficiency}(t_2)$$

The definition of $\geq_T$ is parametric; based on a set P of grouping patterns. Different preferences of reviewers can be captured by changing this set P. It also depends on the part graph g, which is assumed to be implicitly known. By using a partial order, the theory allows for two tours to be incomparable, which is exploited when there is yet no sufficient empirical evidence to base the comparison upon.

The relation $\geq_T$ is defined based on a helper construct, the 'satisfied matches' (SM) of a tour. The satisified matches set consists of all occurrences of a grouping pattern in a tour. A grouping pattern match occurs in a tour when *all* vertices matched for the pattern in the part graph are direct neighbors in the tour. Structurally, a grouping pattern match is a pair (p, v) of a grouping pattern p and the set of matched change parts v.

A tour is better than another when its set of satisfied matches is better, i.e., when it has more matches or the same matches with higher ratings, as given by the grouping pattern's rating function (rate):

$$t_1 \geq_T t_2 \iff \text{sM}(t_1, g) \geq_{SM} \text{sM}(t_2, g)$$
$$\iff \text{sM}(t_1, g) \supset \text{sM}(t_2, g) \vee$$
$$(\text{sM}(t_1, g) = \text{sM}(t_2, g) \wedge \forall m \in \text{sM}(t_1, g) : \text{rate}(m, t_1) \geq \text{rate}(m, t_2))$$

The inclusion of all pattern matches from the sequence of change parts in a tour into the satisfied matches mainly formalizes Principles 1 (group related parts) and 3 (prefer grouping). To

Table 14.6: Relating the formalization of the ordering theory to the empirical findings

Principle/Finding	Way it is accounted for in the formalization
Subjectiveness	By changing the set of patterns, the comparison of tours can be adapted to different preferences or cognitive styles of reviewers.
Principle 1 (group related parts)	A grouping pattern captures the notion of "all related change parts", and the definition of the satisfied matches and the "is better than" relation ensure that in a better tour more related change parts are close together.
Principle 2 (provide information before needed)	It is hard to formalize the notion of provided information; not least because information structures in software are often cyclic, e.g., with the caller of a method providing information on why the callee exists and how it is used and the callee providing information about its pre- and postconditions. Currently, reviewers often resort to heuristics like going bottom-up or top-down along the call flow, and these heuristics can be included in the formalization in the grouping pattern's rating function.
Principle 3 (prefer grouping)	A pattern match is only included in the satisfied matches if the matched parts are close together, and the rating function is only relevant for matches included in the satisfied matches. This is a very strict interpretation of the principle; it was chosen because the participants of the survey rated tours with intervening unrelated change parts low, irrespective of the distance: For 85 of 113 (75%) respondents a tour with one unrelated change part in between was rated as "not very useful" or "not at all useful", and almost the same number said this for two unrelated change parts in between (82 of 113).
Principle 4 (chunking)	The notion of chunking is formalized by the recursive evaluation of sM on shrunk tours and graphs.
Principle 5 (closest is neighbouring)	This principle is more relevant to the presentation of the change parts in the review tool and therefore not explicitly integrated into the formalization.
Principle 6 (understandable rules)	The grouping patterns as a central part of the formalization can be easily explained to software developers. Furthermore, they can be made explicit to the reviewer.
Macro structure	In the study's observations, applying the ordering principles generally leads to a sensible macro structure, too, mostly due to the inclusion of the chunking principle. Therefore, the formalization does not include further measures regarding the macro structure.
Importance order	The importance of a change part for review has not been included in the formalization. Instead, I propose to remove clearly unimportant change parts from the review scope (see Chapter 15) and to optimize the order of the remaining for understandability.
Open questions	There are a number of areas where data is lacking for a grounded formalization, e.g., how to best include differing strengths of gradual relations or differing priorities of grouping patterns. Therefore, a conservative approach is taken and the relation $\geq_T$ is defined to be partial, with the downside that many tours end up as incomparable.

also formalize Principle 4 (chunking), the notion of shrinking a tour (and the corresponding part graph) by combining change parts is introduced: The function $shrink : Tour \times PartGraph \times \mathcal{P}(ChangePart) \rightarrow Tour \times PartGraph$ creates a new tour by removing all change parts contained in the set given as the third parameter and replaces them with a composite part. On the part graph, it also combines all given change parts into the composite part. Edges that pointed to one of the removed parts now point to the composite part. If this leads to duplicate edges or loops, they are combined/removed.

The formalization is concluded by defining the recursive function $sM : Tour \times PartGraph \rightarrow SM$ (pM stands for $patternMatches$, i.e., the matches for a pattern in a tour given a graph):

$$\mathrm{sM}(t, g) := \bigcup_{p \in P} \left(\mathrm{pM}(p, t, g) \cup \bigcup_{m \in \mathrm{pM}(p, t, g)} \mathrm{sM}(\mathrm{shrink}(t, g, m.v)) \right)$$

Table 14.6 shows how the formalization reflects the principles and other empirical findings presented in Section 14.3.

14.5.3 Explanation

To end the presentation of the theory, this section now summarizes and extends its rationale: The theory is based on the assumption that the efficiency and effectiveness of code review (with a fixed number of reviewers) is largely determined by the cognitive processes of the reviewers. The reviewer and the review tool can be regarded as a joint cognitive system, and the efficiency of this system can be improved by off-loading cognitive processes from the reviewer to the tool. The relevant cognitive processes can be divided into two parts: Understanding the code change, and checking for defects. The way in which the changes are presented to the reviewer influences both. A good order helps understanding by reducing the reviewer's cognitive load and by an improved alignment with human cognitive processes (hierarchical chunking and relating). It helps checking for defects by avoiding speculative assumptions and by easing the spotting of inconsistencies.

14.6 An Experiment on Change Part Ordering and Review Efficiency

The ordering theory that was developed in the previous sections is rooted in empirical data. But much of this data is the opinion of developers, and opinions are fallible. A controlled experiment can test the theory more objectively. This section presents the results from such a controlled experiment regarding the impact of change part ordering on review efficiency. The data was gathered together with the data on working memory, cognitive load, and review performance that is presented in Chapter 13. The current section presents the remaing details of the experiment design and the respective results.

$\mathrm{RQ}_{14.4}$ asks: Can the order of presenting code change parts to the reviewer influence code review efficiency, and does this depend on working memory capacity? Suppose there are two orders a and b for a given code change and that $a \geq_T b$. Then the proposed theory predicts that review efficiency and effectiveness for a is not inferior to that of b. For efficiency it also predicts that there are cases where efficiency is greater, based on the rationale that a better order might lead to a faster review with the same found defects, or to finding more defects in

Table 14.7: Considered order types

ID	Origin	Explanation
OF	'optimal + files'	a best order (i.e., a maximal element according to $\geq_T$) that keeps file boundaries intact
ONF	'optimal + no files'	a best order (i.e., a maximal element according to $\geq_T$) that is allowed to ignore file boundaries
WF	'worst + files'	a worst order (i.e., a minimal element according to $\geq_T$) that keeps file boundaries intact
WNF	'worst + no files'	a worst order (i.e., a minimal element according to $\geq_T$) that is allowed to ignore file boundaries

the same amount of time. Focusing on efficiency, this leads to the following null and alternative hypotheses:[1]

$H_{3.<a,b>.0}$ reviewEfficiency(a) = reviewEfficiency(b)

$H_{3.<a,b>.A}$ reviewEfficiency(a) $\neq$ reviewEfficiency(b)

To study the dependence on working memory capacity, the sub-sample with less than median working memory capacity is also tested. This sub-sample should be more prone to cognitive overload.

For a given code change of non-trivial size, there is a vast number of possible permutations and consequently many possible comparisons to perform; a subset has to be selected for the experiment. As the experiment is the first to measure the effect of change part ordering on code review, the choice is exploratory: The experiment compares one of the best change part orders according to $\geq_T$ with one of the worst orders. Usually, such a worst order mixes change parts from different files. As a more realistic comparison, one of the best and one of the worst orders that keep change parts from the same file together are also included. In the following, these order types are called as described by the terms in the column *ID* in Table 14.7.

By construction, it holds that $OF >_T WF$, $ONF >_T WNF$, $WF >_T WNF$ and, by transitivity, $OF >_T WNF$. The experiment considers the first 3 pairs. When inserted into the above-mentioned hypotheses, they give rise to a total of 3 combinations of null and alternative hypotheses, each named after the first order in the pair: $H_{3.OF.0}$, $H_{3.OF.A}$, $H_{3.ONF.0}$, ...

14.6.1   Design

The general structure follows a partially counter-balanced repeated measures design [123], augmented with additional phases. It is 'partially counter-balanced' because the order of treatments and patches is randomized, but only two of the four treatments per participant are measured. The basic experimental setup was already described in Section 13.1.

When designing an experiment on code reviews, one has to account for large variations in the review performance between participants. Therefore, counterbalanced repeated-measures designs are common (as argued, for example, by Laitenberger [215]). To gain maximum information from the experiment, it would be desirable to gather data for each of the four types of orders from each participant. But this is infeasible due to the large amount of participants' time and effort needed for each review. Therefore, we (Baum, Bacchelli) decided to restrict ourselves to pairs of change part order types, specifically, those pairs needed for checking the predictions

---

[1]The two-sided formulation is used for reasons of conservatism, even though the theory's prediction is one-sided

for $RQ_{14.4}$: ONF vs WNF, OF vs WF and WF vs WNF. Each participant is shown a different change in each review. For each pair of order type, the experiment uses a fully counter-balanced design. Consequently, there are four groups per pair, differing in the orders of change part order type and of code change.

To determine the four different orders (ONF, OF, WNF, WF) for each of the two code changes, I first split the code changes into change parts. I mainly split along method boundaries, i.e., if several parts of the same method were changed, the method was kept intact and regarded as a single change part. If a whole class was added, it was kept intact. After that, I determined the relations between the change parts. I checked for the following subset of the relation types identified from the ordering interviews: (1) Similarity (moved code or Jaccard similarity [182] of used tokens > 0.7), (2) declare & use, (3) class hierarchy, (4) call flow, and (5) file order. The previous sections did not specify which relation types should be regarded as more important than others. To determine a concrete order, I had to assume a certain priority and used the order just given (i.e., similarity as most important; file order as least important). To construct the OF and WF orders, an additional relation type 'in same file' was added as the top priority. To find the orders based on the relations, I implemented the partial order $\geq_T$ in software and semi-automatically constructed minimal and maximal elements of this partial order relation.

One of the experiment's main variables of interest is review efficiency, measured as the ratio of found defects and needed time. It is generally believed that there is a trade-off between speed and quality in reviews [140], which lets many researchers control for time by fixing it to a certain amount. This would run contrary to one of the experiment's main research goals, i.e., finding differences in efficiency. Therefore, participants were allowed to review as long as they deem it necessary and the total time was measured. A participant who needed to interrupt the review could press a 'pause' button; 14 participants did so at least once. The study measures gross time (including pauses) and net time (without pauses).

I performed power analysis for $RQ_{14.4}$, because I expected a smaller effect for it and it had more groups compared to the other RQs in the experiment (Chapter 13). I used Monte Carlo power analysis: I implemented a simple simulation model for the experiment[2], estimated effect sizes and some parameters and performed randomized simulation runs with and without a simulated effect. Based on 1000 simulated experiments per run, the model estimated false positive and false negative rates. Among the analyzed variants were different choices for the treatment groups, different statistical tests and different sample sizes. We settled for the incomplete repeated-measures design described so far. To deal with this design and allow for potentially imbalanced groups and several confounding factors, we planned to use linear mixed effect (lme) regression models [26] and determine confidence intervals for the coefficients using parameterized bootstrap. The dependent variable is review efficiency ('Number of detected defects' / 'Needed net review time'), independent variables are 'used change part order type' and 'first or second large review' (fixed effects) and 'used code change' and participant ID (random effects). Using further of the measured confounders in the model is not beneficial here as they are subsumed in the random effect per participant. It turned out that some of the assumptions in the simulation were wrong, most notably that it did not take drop-outs into account. Also, the empirical data does not satisfy all assumptions needed for lme models.[3] A non-parametric alternative with a minimal set of assumptions is the Wilcoxon signed-rank test, but it is also problematic due to imbalanced groups after removal of drop-outs. Therefore, both results are presented to the

---

[2]'generateTestData.r' and 'SimulateExperiment.java' in the replication package [42]

[3]for further discussion, see 'Statistical Conclusion Validity' in Section 14.7

Table 14.8: Number of participants by treatment groups, with details for order of treatment and order of code change. Only the first large review is given for each group, the second review is the respective other value (e.g., in the OF-WF group, when WF+Change A was reviewed first by a participant, OF+Change B was second).

	OF-WF			ONF-WNF			WF-WNF		
	OF first	WF first	total	ONF first	WNF first	total	WF first	WNF first	total
Change A first	5	6	11	5	4	9	3	2	5
Change B first	3	4	7	4	5	9	4	5	9
total	8	10	18	9	9	18	7	7	14[1]

[1] The slightly lower number for the WF-WNF combinations is intended, the balancing algorithm slightly favored the other two treatment combinations.

reader. The R code of all analyses is available [42]. The full set of all participants as well as the subset of participants with lower than median working memory capacity is analyzed, but the difference is not tested formally.

14.6.2 Results

Due to the online assignment of participants to groups and due to data cleansing, the number of participants per group is not fully balanced, especially in the WF-WNF group. Table 14.8 shows the distribution. There are also signs that the review skill is not equally distributed among the three groups: The mean efficiencies in the small, warm-up review are 13.9 (OF-WF), 17.23 (ONF-WNF), and 15.88 defects/hour (WF-WNF).

$RQ_{14.4}$ asks: "Can the order of presenting code change parts to the reviewer influence code review efficiency, and does this depend on working memory capacity?" Figure 14.5 shows box plots with efficiency for the different change part orders for each of the treatment groups. The difference in medians is in the direction predicted by theory, but subject to a lot of variation (Table 14.9 shows the exact numbers). As mentioned in the experiment design, it was originally planned to analyze the full data with a linear mixed effect model [26]. Such a model can take the slightly unbalanced and incomplete nature of the data into account, but its preconditions (independence, homoscedasticity, normality of residuals) were not fully satisfied. To ensure independence, each treatment group is analyzed separately instead of being pooled. Regarding the other preconditions, the data is also analyzed with a Wilcoxon signed-rank test, acknowledging that it is more prone to bias due to ordering and/or different code changes. With neither analysis, there is an effect that is statistically significant at the 5% level (especially after taking alpha error correction into account). Looking at the tendencies in the data, there seems to be a medium-sized positive effect of the OF order compared to WF. For ONF vs WNF instead, there is a conflict in the direction of the effect between mean and median. As expected, the difference between WF and WNF is small. The theory suggests that the positive effect of a better ordering should be larger for participants with lesser working memory capacity. Except for the ONF-WNF sub-sample, the tendencies for the respective sub-samples in Table 14.9 support that prediction, but the sub-samples are too small to draw meaningful statistical conclusions.

For the percentage of understanding questions answered correctly at the end of the reviews, the results are weaker but otherwise similar: Largely compatible tendencies but statistically
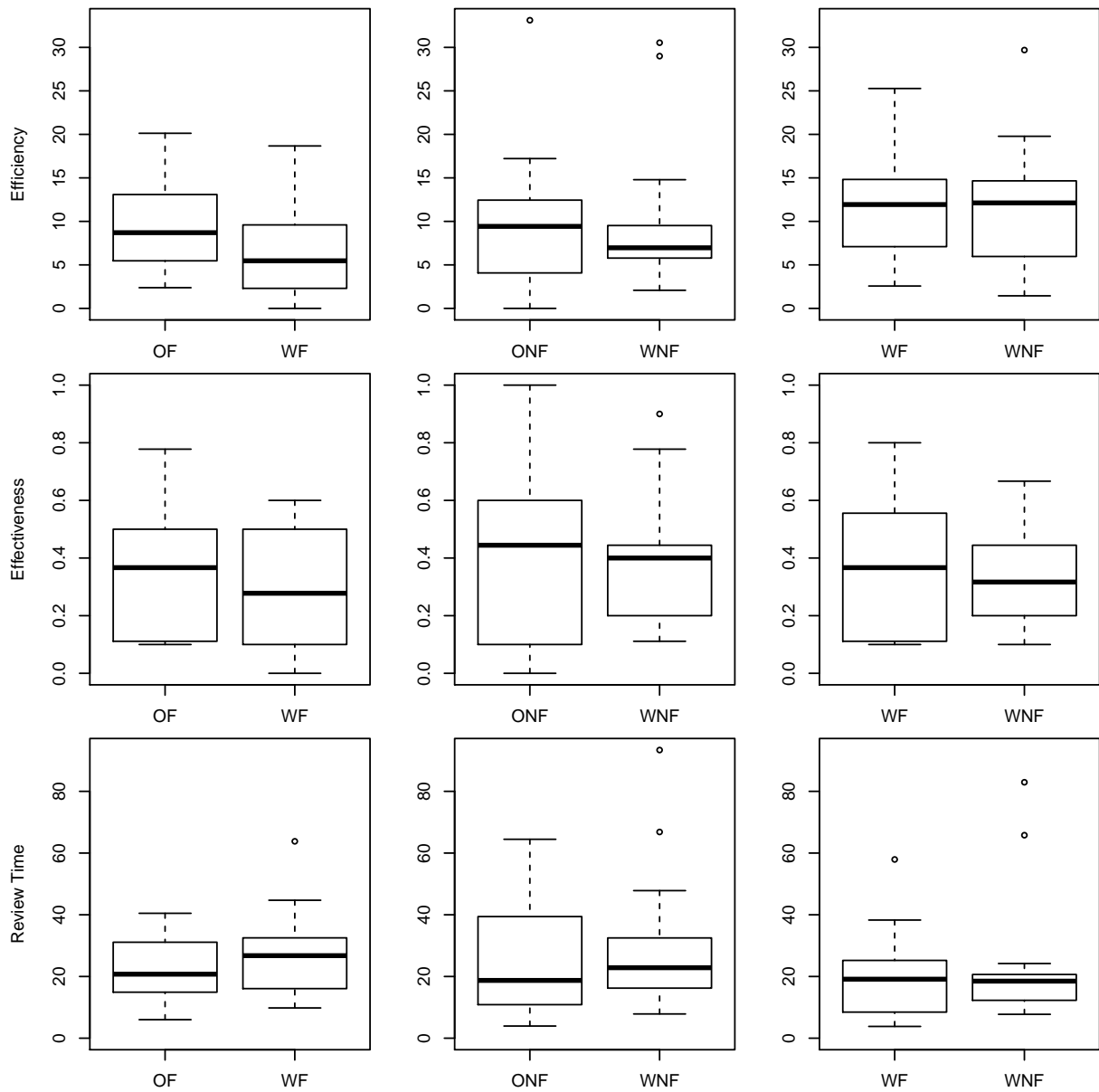
Figure 14.5: Box plots for review efficiency (in defects/hour), effectiveness (found defects/total defects), and review time (in minutes) for the three treatment groups. In each plot, the left treatment is the theoretically better one.

Table 14.9: Comparison of efficiency (in defects/hour) for the different change part orders; overall, for each treatment combination and for the subsamples with below median working memory capacity. Caution has to be applied when interpreting the results of lmer as not all assumptions are met. Due to the small samples, lmer models for low wm span are left out and the respective intervals are inaccurate. Every row from the upper part is continued in the lower part. 'conf.int.' = 'confidence interval', 'sd' = 'standard deviation', 'negl.' = 'negligible'

group	n	theoretically better treatment median (conf.int.)	mean (sd)	theoretically worse treatment median (conf.int.)	mean (sd)
all	50	10.1 (7.1 .. 11.4)	9.9 (6.2)	7.0 (5.8 .. 9.3)	9.1 (7.0)
low wm span	22	11.9 (6.5 .. 13.4)	11.2 (5.9)	7.2 (5.3 .. 9.6)	9.8 (8.2)
OF-WF	18	8.7 (5.5 .. 13.0)	9.1 (4.7)	5.5 (2.3 .. 8.1)	6.3 (5.0)
OF-WF, low	11	10.1 (3.9 .. 13.4)	10.2 (5.1)	5.3 (0.0 .. 8.1)	5.8 (5.4)
ONF-WNF	18	9.4 (4.1 .. 10.8)	9.4 (7.8)	7.0 (5.8 .. 9.1)	9.8 (7.8)
ONF-WNF, low	6	8.6 (0.0 .. 12.4)	8.7 (5.9)	7.2 (6.3 .. 9.5)	11.0 (8.9)
WF-WNF	14	11.9 (5.6 .. 14.8)	11.4 (5.8)	12.1 (5.5 .. 14.7)	11.8 (7.4)
WF-WNF, low	5	15.7 (11.4 .. 25.3)	16.4 (5.3)	15.5 (6.0 .. 29.7)	17.0 (8.7)

group	comparison relative difference of medians	p (Wilcoxon)	Cliff's δ	lmer coefficient (conf. int.)[1]
all	44%	0.2587	0.14 (negl.)	
low wm span	67%	0.2479	0.22 (small)	
OF-WF	59%	0.1084	0.35 (medium)	2.66 (-0.66 .. 5.45)
OF-WF, low	91%	0.0537	0.52 (large)	–
ONF-WNF	35%	1.0000	0.07 (negl.)	-0.36 (-2.63 .. 2.04)
ONF-WNF, low	21%	0.5625	0 (negl.)	–
WF-WNF	-2%	0.6698	0.02 (negl.)	-0.38 (-3.56 .. 3.15)
WF-WNF, low	1%	0.6250	-0.04 (negl.)	–

[1] Goodness of fit for the lmer models [22]: OF-WF $R_m^2 = 0.11$, $R_c^2 = 0.17$; ONF-WNF $R_m^2 = 0.01$, $R_c^2 = 0.81$; WF-WNF $R_m^2 = 0.03$, $R_c^2 = 0.51$

non-significant results, with the strongest effect for the OF-WF condition. Mean correctness in detail: OF-WF 67.1% vs 56.5%, ONF-WNF 60.6% vs 64.8%, WF-WNF 63.1% vs 60.1%.

Part of the hypothesis underlying RQ$_{14.4}$ is that a better order means less mental load. To roughly assess mental load, the experiment UI asked participants to rate which of the large changes was subjectively more complicated. In line with the hypothesis, a majority of the participants rated the worse order as more complicated (answer as expected: 26 participants, no difference: 11, inverse: 13). Interestingly, this difference does not carry through to subjective differences in understanding (as expected: 22, no difference: 10, inverse: 18). When justifying their choice when comparing the two large reviews for complexity and understanding, many participants gave reasons based on properties of the code changes (e.g., *"The change in the second review contained a more behavioral change whereas the change in the third review involved*

a more structural change"$_{PC5}$). But many of their explanations can also be attributed to the different change part orders, *e.g.*: *"Changes in the same files were distributed across changes at various positions"$_{PC4}$*, *"the [theoretically worse review] involved moving of much code which was hard to track"$_{PC1}$*, *"there were jumps between algorithm and implementation parts for the [theoretically worse review]"$_{PO6}$*, *"in the [theoretically worse] review changes within one file were not presented in the order in which they appear in the file."$_{PO11}$*, *"The [theoretically worse review] was pure code chaos. [The theoretically better one] was at least a bit more ordered."$_{PC8}$* or plainly *"very confusing"$_{PC6}$*.

The theory proposed in Section 14.5 contains additional hypotheses that a better change part order will not lead to worse review efficiency or effectiveness. As the examination of the box plots in Figure 14.5 and the confidence intervals in Table 14.9 gives little reason to expect statistically significant support or rejection, formal non-inferiority tests are left out.

> **RQ**$_{14.4}$: *Strong statistically significant conclusions cannot be drawn. If an effect of better change part ordering on review efficiency exists, its strength is highly dependent on the respective orders.*

14.7 Validity and Limitations

This section presents limitations for the results on change part ordering for reviews. The *theory-generating* part of the study is limited by two assumptions underlying the argumentation: (1) For the interviews and survey: What *is* a good tour and what experienced developers *think* is a good tour coincides to a large degree. (2) For the usage of related work: Findings from natural language reading and program comprehension can be transferred to code change comprehension. To overcome these and other limitations, the results from the various data sources were triangulated, as detailed in Table 14.10. Most importantly, the theory was tested with the controlled experiment described in Section 14.6. The following sections give more details on limitations of the used data sources.

Log Data. The main limitation of the collected log and repository data of Section 14.2 is that

Table 14.10: Triangulation of the single data source's weaknesses with a mixed-methods approach

Study	Weakness	Triangulated by
Log Data	• single company	→ survey
	• no data on goodness of order	→ interview, survey, experiment
Interviews	• small sample	→ survey, literature
	• researcher bias	→ survey, experiment
	• depends on opinion of developers	→ literature, experiment
Literature Study	• not specific for code order in reviews	→ interviews
Survey	• depends on opinion of developers only	→ experiment
	• needs to focus on few specific hypotheses	→ interviews, literature
Experiment	• depends on opinion of developers only	→ experiment
	• needs to focus on few specific hypotheses	→ interviews, literature
	• missing statistical significance	–

it comes only from a single company and code review tool. Since the general tendency in this data was also supported in the survey, this should only be a limitation to the generalizability of the exact numbers.

Task-Guided Interviews and Survey. One of the greatest risks in the ordering interviews and the survey was to accidentally introduce a bias for a certain order. Several measures were taken to counter this risk: In the interviews, all change parts were shuffled and random words instead of numbers were used as IDs for the distinct parts. Line numbers were not removed as they could be part of a sensible ordering strategy. The survey used randomized orders in the questions and in the descriptions, and used colors as part IDs. Color names instead of random words were found to be easier to understand in the pre-tests. To avoid the anchoring effect [382] in both the interviews and survey, the interviewers first asked for the participant's preferred order before presenting other orders. Describing the steps of the interview in an interview guide and videotaping some sessions increased intersubjectivity. Mitigating researcher bias was one of the reasons to perform a joint card sort.

During survey creation, I checked against published guidelines [183] and we (Baum, Bacchelli) performed several rounds of pre-testing. Nevertheless two factors probably introduced some noise into the data: (1) Understanding the abstract situations was still a problem for some participants (some respondents indicated that they left the respective questions empty, but others might have answered without having understood the described situation); (2) the drag and drop support in the ranking widget (used for the questions of type Point 3 in Figure 14.2) had to be used with a certain care to avoid unintended results.

A negative side-effect of the sampling method is that the sample should be regarded as self-selected; the survey included a number of questions to characterize the sample and check for influencing factors.

The generalizability of the survey results to the population of users of change-based review is probably quite high, mainly due to the large number of participants in the survey. The sample of distinct code changes in the survey is much smaller, which could impede generalizability in this regard. Specifically, the set of relations given in Section 14.3 may be incomplete.

Controlled Experiment. Many general limitations of the experiment were already discussed in Section 13.3. What follows is a discussion more specific to the ordering-related parts of the experiment.

One of the central measures needed to determine review efficiency is the time taken for a review. By having a 'pause' button and measuring time with and without pauses, the experiment UI allowed to measure time more accurately, but it cannot be assured that all participants used this button as intended. A risk when measuring time is that the total time allotted for the experiment, which was known to the participants, could bias their review speed. To partially counter this threat, participants were not told the number of the review tasks, so that the time allotted per task was unknown to them. To avoid one participant's time influencing another's in the lab setting, only one participant took part at a time. Participants were asked to not talk to others about details of the experiment but it is not known whether all complied. Participants did not get feedback on their review performance during the experiment, to not influence them to go faster or more thoroughly than they normally would. A common threat in software engineering studies is hypothesis guessing by the participants. The experiment UI stated only abstractly that we are interested in improving the efficiency of code review and did not mention ordering of code at all.

A sample of 50 professional software developers is large in comparison to many experiments in software engineering [351]. For other sources of variation, the experiment was limited to con-

siderably smaller samples, leading to a risk of mono-operation bias and limited generalizability: There are only two large code changes, and only four different change part orders.

A likely consequence of the difficult task is the observed ordering effect, i.e., participants spent less time on the last review. By random, balanced group assignment and inclusion of the review number in the regression model for $RQ_{14.4}$ we tried to counter the ensuing risk. Due to drop-outs and the failed assumptions of the lme model we did not fully succeed. There is a high drop-out rate, likely again a consequence of the difficult task and the online setting. A larger share of drop-outs, 23 of 37, happened when either a WF or WNF order was shown. On average, the drop-outs have lower review practice and performed worse in the short review (differential attrition), which could partly explain the differences between groups described in Section 14.6.2. Each treatment group had to be analyzed separately to counter that risk.

A failure to reach statistically significant results is problematic because it can have multiple causes, e.g., a non-existent or too small effect or a too small sample size. Based on the power analysis for $RQ_{14.4}$, we planned to reach a larger sample of participants than we finally got, and the sample we got had to be split into three sub-samples due to differential attrition. This could be a reason that statistical significance is not reached for $RQ_{14.4}$. For the analysis with linear-mixed effect models for $RQ_{14.4}$, the statistical conclusions could be influenced by failing to meet several assumptions of these models (normality of residuals, homoscedasticity). Therefore, we decided to also report results from Wilcoxon signed-rank tests, which do not share these assumptions. But these tests have their own problems, mainly that they do not account for the imbalance in treatment group sizes and have lower power. All in all, it is not possible to draw reliable conclusions for $RQ_{14.4}$.

$$\Diamond$$

This chapter went a long way, from motivating why improved code ordering for code reviews is worthwhile, over inductive generation of a theory for a good ordering, to a controlled experiment that tests this theory. An efficient algorithm for the ordering is now implemented in CoRT. Overall, the results are promising, but unfortunately the controlled experiment did not give conclusive results.

15

Classification of Change Parts

This chapter studies how the computer can support the human reviewer by automatically analyzing and classifying the change parts to review. Marking change parts as irrelevant for review is derived as a promising possibility for such support, extending the argument from Chapter 12. An approach that uses data extracted from software repositories to create such a classification model is presented. It is based on [31], joint work with Steffen Herbold and Kurt Schneider.

In sum, the purpose of this chapter's study is to analyze ways to use repository mining to identify the importance of change parts for code review and to improve code review efficiency based on this information. This chapter includes a discussion on the foundations of the approach, rooted in research on code reviews and, to some degree, cognitive psychology. The resulting concepts are applied in a case study at the partner company. The study's ambition is to go the whole way from the initial discussion to the use in practice. By using an interactive and multi-objective approach for rule mining, the study leaves the beaten path followed by most of the current defect prediction research to explore a promising alternative.

As motivated in Chapter 12, the focus is on support for the reviewer. Answering this chapter's first research question derives possibilities to help the reviewer, assuming that the importance of the parts of the code change under review is known:

> **RQ$_{15.1}$.** How can information on change part importance help to reach code review goals more efficiently? What is "importance" in this regard?

RQ$_{15.1}$ is treated deductively, based on earlier findings from this thesis and others.

A model to determine change part importance can be built based on empirical data that shows which change parts act as triggers for review remarks. We (Baum, Herbold) extract such empirical data from the partner company's source code repository and use the data to build a prediction model. We choose a mining approach based on preferences articulated by the developers at the partner company. We evaluate the results of our approach and a standard rule mining approach, both subjectively with the developers and objectively based on performance metrics:

> **RQ**$_{15.2.1}$**.** Which requirements for the classification model are considered most important by the developers in the partner company?
> **RQ**$_{15.2.2}$**.** What are the characteristics of good rulesets found in the data?
> **RQ**$_{15.2.3}$**.** How good are the found rulesets in the developers' opinions?
> **RQ**$_{15.2.4}$**.** How well do the found rulesets perform on unseen data?

15.1 Methodology

For RQ$_{15.1}$, a purely deductive methodology is used: Based on findings from earlier chapters, on concepts from cognitive psychology, and on logical arguments, possibilities to consider change part importance in code reviews are discussed and evaluated.

The remaining research questions in this chapter are assessed with a case study, which studies review remark prediction in the context of the partner company. The research design is flexible, and it pragmatically mixes methods: The study combines qualitative as well as quantitative data, e.g., when triangulating the quality of the found rules from the results on the extracted data (Sections 15.4.3 and 15.4.5) and opinions from the team (Section 15.4.4). The specific data sources are described in the sections where they are used.

Both case study research, as well as data mining, are highly iterative endeavors [250, 328]. I also used an iterative approach, starting with the initial stages of the study in fall 2017, until the discussion of the study's results with the development team in fall 2018. Figure 15.1 depicts central data sources and research steps. The current chapter linearizes the results. It starts with the goal of the mining process (Section 15.2), followed by the approach for data extraction and mining (Section 15.3). Finally, the results of applying the approach to the company data are shown in Section 15.4. The chapter concludes by discussing problems (Section 15.5), threats to validity (Section 15.6) and related work (Section 15.7).
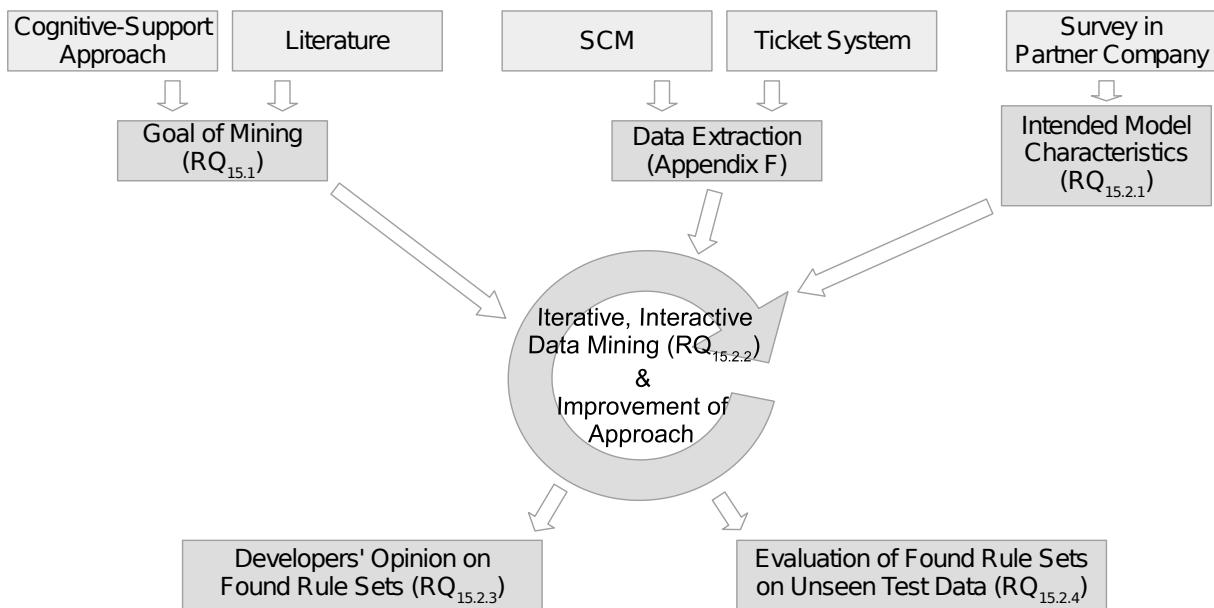
Figure 15.1: Overview of the data sources and steps used in this chapter

15.2 Use of Change Part Classification to Reach Code Review Goals more Efficiently

15.2.1 Possibilities for Using Change Part Importance to Improve Review

In other studies, defect prediction or static analysis are used to create *review agents* (e.g., [69, 233]). Those agents act as a computerized reviewer by highlighting certain parts of the changed code as problematic, with varying degrees of specificity. The results of the agents are then either meant to be double-checked by a human reviewer, or they are communicated directly to the author. Depending on the quality of the review agent, these approaches can be useful, but there are further possibilities for support. Therefore, another point of view to use change part classification to improve review performance is proposed here. The proposed approach follows the agenda of cognitive-support code review tools by focusing on support for the reviewer during checking and by relying on theories of human cognitive processing.

Rasmussen [311] put forward that it is beneficial to regard human cognitive processing to happen on one of three levels: (1) Skill-based behavior "*take[s] place without conscious control as smooth, automated, and highly integrated patterns of behavior.*" Examples of skill-based behavior are bicycle riding or musical performance (for experienced cyclists resp. musicians). (2) Rule-based behavior is "*controlled by a stored rule or procedure*", which has been acquired in the past and which is activated based upon the given situation. In contrast to skill-based behavior, "*the higher level rule-based coordination is generally based on explicit know-how, and the rules used can be reported by the person*". He gives an example of a skilled operator who observes some measurements and controls valves accordingly. (3) Knowledge-based behavior is characterized by effortful explicit mental processing, and "*the internal structure of the system is explicitly represented by a "mental model" which may take several different forms*". It occurs "*during unfamiliar situations, faced with an environment for which no know-how or rules for control are available from previous encounters*". After the first letters of the three levels, Rasmussen's model is called 'SRK taxonomy'.

Checking of the source code during a review can also happen on each of Rasmussen's levels. When trying to understand an unfamiliar algorithm, the reviewer is acting on the knowledge-based level. An example of rule-based processing is when an experienced reviewer looks at the code and finds a pattern of 'array access' that triggers the rule 'there should be boundary checks'. Skill-based checking in a review is probably rare; an example could be finding a typographic
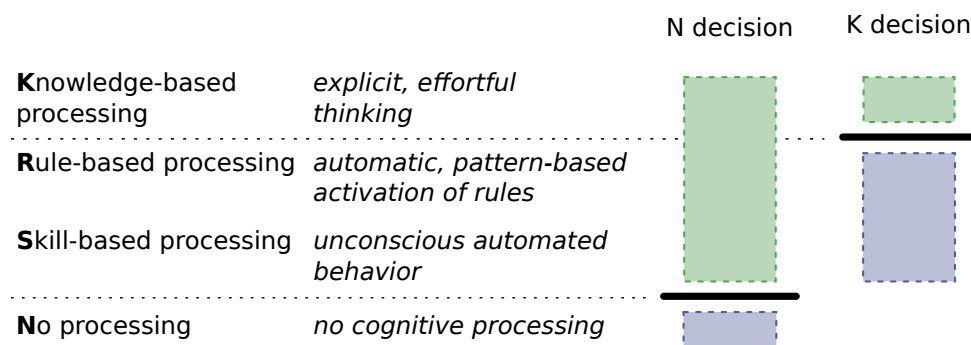
Figure 15.2: The SRK taxonomy with 'no processing' added, and the two options to decide how to process a change-part

Table 15.1: Pros and cons for the possibilities to use the information on change-part importance for reviewing

Option	Pros	Cons
(1) Show importance value	Most information available for user	Might not reduce effort for decision
(2) Sort by importance	Ensures that most important parts have been read when review is ended prematurely	Resulting order can be confusing to the reviewer; Does not provide a clue for the reviewer whether and when the review should be ended early
(3) Leave out unimportant	Most reduction in effort for reviewer	Hard threshold needed

error. Rasmussen's model assumes that some cognitive processing is done by the human. In the following, it will be useful to introduce the distinction between processing and no processing explicitly as a fourth level ('no processing'). This happens, for example, when the reviewer decides to skip the review of certain parts of the changes. One of the core assumptions for the remaining chapter is that the most efficient cognitive mode can differ between the change parts in a review. For some parts, it is worth to think deeply about them, whereas it is sufficient for others to skim them (i.e., to process them rule-based and skill-based only) or not have a look at them at all. In the interviews of Part I and the ordering study described in Chapter 14, some of the developers stated accordingly that not all parts of an artifact under review are equally important for the review. Varying importance for review has also been mentioned in much older works, e.g., by Gilb and Graham [140]. To some degree, the cognitive mode of checking a change part can be chosen (Figure 15.2): The reviewer can decide to not expend the effort to do knowledge-based processing and stay on the skill- and rule-based level ('K decision'), and the reviewer can decide to not check a certain part at all ('N decision'). This chapter proposes to provide computer support for this choice of mode. In the following, the study focuses on the N decision and leaves the K decision for future work, for three reasons: (1) There is more existing support that not reviewing certain change parts at all will help to increase review performance both directly by saving time and indirectly by reducing the cognitive load (see Chapters 12 and 13). (2) Relative to the checking itself, the effort for the K decision is much smaller than for the N decision. Therefore, a model for the N decision is already beneficial when it has a performance similar to the human, whereas a model for the K decision needs to be better than the unsupported reviewer's choice. (3) Data to learn a model for the N decision can more readily be extracted from software repositories.

So the goal for the remaining chapter is to support the reviewer when deciding whether to skip certain change parts during a review. A change part can be skipped when its inclusion is not important for one of the review goals. Section 15.2.2 discusses what this means in detail. Given that such an importance value is known, it can be used in at least three ways: (1) The importance value can be presented to the reviewer so that he or she can decide whether to review. (2) The change parts can be sorted from most important to least important. (3) The review tool can leave out all change parts less important than a certain threshold from the proposed review tour. Table 15.1 shows pros and cons of these options. Because it has the highest potential to reduce cognitive load and does not interfere with the optimal code order as determined in Chapter 14, option 3 is pursued in the following.

Table 15.2: Assessment of the effect of leaving out change parts from reviews on the attainment of review goals and avoidance of unintended review side-effects.

Review goals:
Better code quality. The direct positive effect of code reviews on code quality is based on the correction of the code according to the review remarks. Reaching this goal is unaffected as long as the review of the reduced set of change parts still leads to the same fixed remarks.
Finding defects. Same as above, reaching this goal depends on the fixed remarks.
Finding better solutions. When the review discussion leads to better solutions, this results in changes to the code base (i.e., fixed review remarks), so an indicator that this goal is still reached is again when the fixed remarks stay the same.
Learning (author). The code's author mainly learns from the review remarks communicated to him/her, so that this goal will be unaffected when the communicated remarks stay the same.
Learning (reviewer). This goal is unaffected when the knowledge gained from the shrunk review set is comparable to the original review. A necessary (but not sufficient) precondition for that is that the shrunk change under review is still understandable.
Complying to QA guidelines. When the quality assurance guidelines demand the review of certain parts of the code, these may not be left out from the review scope.
Improved sense of mutual responsibility. The attainment of this goal mainly depends on doing reviews at all, not on the exact review scope.
Team awareness. The attainment of this goal mainly depends on how the reviews are communicated, not on the exact review scope.
Track rationale. Review remarks can point to portions of the code with an insufficient description of the rationale, so the attainment of this goal also depends on the set of communicated review remarks.
Avoid build breaks. Usually, build breaks can be avoided more efficiently by automatic checks than by manual reviews. In case reviews are still used for this goal, its attainment also depends on the set of fixed remarks.
Review side-effects:
Avoid higher staff effort Leaving out parts to review should usually lead to lower review effort and, therefore, help to attain this goal. A potentially adverse effect could occur when the understandability of the change is hampered by leaving out change parts needed for understanding.
Avoid increased cycle time Same as above, the review duration could only be negatively affected when the understandability of the change is hampered.
Avoid offending/social problems Leaving out change parts based on objective rules probably does not influence the risk of offending people or provoking other social problems through reviews.

15.2.2 Influence of Leaving out Change Parts on Possible Review Goals and Derivation of Target Metrics

Table 15.2 shows for each of the common review goals how leaving out change parts can make a difference in achieving the goal. The goals have been consolidated from Section 4.3 and Bacchelli and Bird's findings [18]. The essence of the analysis is that most of the goals depend either on the communicated or the fixed review remarks. The decision whether to communicate a remark or to fix it on-the-fly is unlikely to be affected by shrinking the review scope so that these are treated the same in the following.

There are two main preconditions for a reviewer to find a remark: The reviewer needs to understand the relevant portions of the code and needs to observe a *trigger* for the remarks. Consequently, the set of review remarks stays the same when the relevant code portions are still understandable and when there is still at least one trigger for each review remark in the shrunk

change. The notion of triggers for review remarks is discussed further in Section 15.3.1.

Two goals do not directly relate to review remarks: 'complying to QA guidelines' and 'learning of the reviewer'. For many teams, these are the least important goals (see Section 4.3), whereas 'better code quality' and 'finding defects' are very often among the top reasons for doing reviews. Therefore, the study focuses on shrinking the review scope while leaving the set of triggered remarks and the understandability of the change intact.

In commercial software development, code reviews are usually not an end in itself but are a means among several to obtain a high profit or return on investment. Therefore, it is often acceptable to find a few review remarks less when this is offset by higher savings in the review effort. A simple model for the profit of shrinking the changeset is:

$$
\begin{aligned}
\text{profit}_{shrinking} &= \text{savings} - \text{effort}_{missedRemarks} \\
&= \sum_{l \in \text{leftOut}} \text{reviewEffort}(l) - \sum_{m \in \text{missedRemarks}} \text{cost}(m)
\end{aligned}
\tag{15.1}
$$

As a further simplification, a constant cost factor c that relates the review effort for change parts to the effort caused by missed remarks is assumed, i.e., $profit \propto |leftOut| - c \cdot |missedRemarks|$. The time needed to review a change part can be estimated based on the empirical data from code review logs, but the effort caused by a missed review remark is much harder to quantify. The study uses two alternatives: (1) The cost factor is treated as unknown and the break-even point for a positive profit calculated, i.e., the maximum cost per missed remark up to which the profit will still be positive for a given rule. A rule with a higher break-even point is more conservative and therefore usually better. (2) The mining UI (see Section 15.3) provides the results for several cost factors to the user so that he or she can easily compare the possibilities.

> **RQ**$_{15.1}$: *Information on change part importance can help the reviewer to reach code review goals more efficiently by: (1) Leaving out change parts that neither trigger review remarks nor are needed for understanding, or (2) highlighting change parts that the reviewer should review in a higher cognitive mode than he or she otherwise would. The rest of the chapter focuses on (1) and takes the point of view that a slight decrease in found remarks is acceptable when offset by a significant saving in effort, i.e., when the overall profit is positive.*

15.3 Approach for Data Extraction and Model Creation

Next, an overview of how the data for the mining study was extracted from the partner company's repositories and used to build a prediction model is given.

15.3.1 Extracting Potential Triggers for Review Remarks from Repositories

This section motivates the approach taken to extract the needed data from the partner company's SCM and ticket system. A more detailed discussion of the rationale and applied simplifications, and a description of the used algorithms, can be found in Appendix E.

As discussed in the previous section, a change part is not important for review when it does not lead to a review remark (i.e., it is not a 'trigger for a remark') and is also not needed to understand a trigger. Multiple change parts can act as a trigger for the same review remark, for
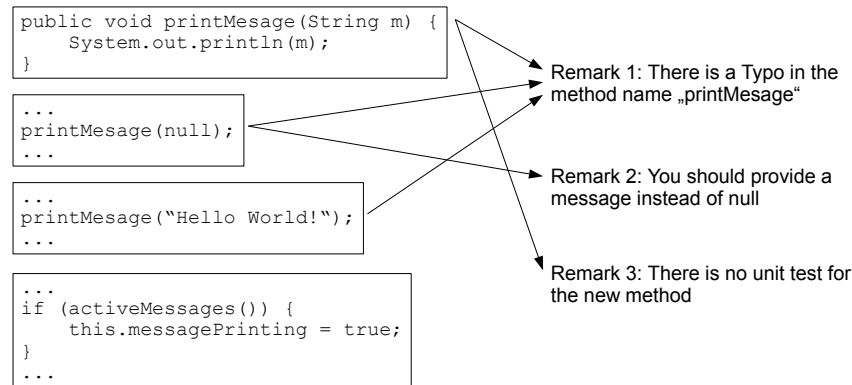
Figure 15.3: Example how change parts can act as triggers for review remarks. Arrows mean 'can be trigger for'.

example, when a problem in a method's interface is manifested in several calls to that method. Furthermore, a change part can also contain multiple problems, i.e., be a trigger for multiple remarks. Figure 15.3 shows an example with several change parts and remarks.

In a very simple situation, each review would deal with only one commit and the used review tool would store which change parts from that commit triggered each remark. The situation in the partner company is more complicated in several ways:

- Structured information on remark positions is only available for a fraction of the reviews. CoRT stores the position of review remarks, but reviews before its introduction don't have that information, as do reviews performed by non-users of CoRT. Therefore, the actual changes performed in reaction to the review are extracted from the company's SCM and used as a proxy for review remarks.

- Remark positions refer to the state of the code base at the time of review. The extracted review remarks are from a later state of the code base than the commits under review. Therefore, they need to be traced back to the original commits. This is similar to the tracing of defects to defect-introducing commits with the SZZ approach [353], but it is not the same. This thesis proposes an adjusted tracing algorithm, called RRT. It can be found in Appendix E.4, and Appendix E.5 shows how much RRT's results differ from those of simply applying SZZ.

- Reviews are performed for multiple commits at once. This can lead to situations where the same part of the code was changed multiple times so that the tracing algorithm cannot tell which of these is the correct trigger.

- There are commits that cannot be triggers for a remark. As the company performs post-commit reviews, other changes outside of the reviewed ticket can happen to the codebase. Because these do not change the review scope, it would be useless to treat them as change parts that can be left out from the review scope, and they need to be skipped when tracing remarks to their triggers.

Figure 15.4 exemplifies a situation in which a review remark is traced back to potential triggers. The mentioned complications are not specific to the partner company. For example, Part I's survey[1] revealed that 33% of the teams do not use a specialized code review tool and that 54% of the teams perform post-commit reviews.

---

[1]with detailed numbers in Appendix B
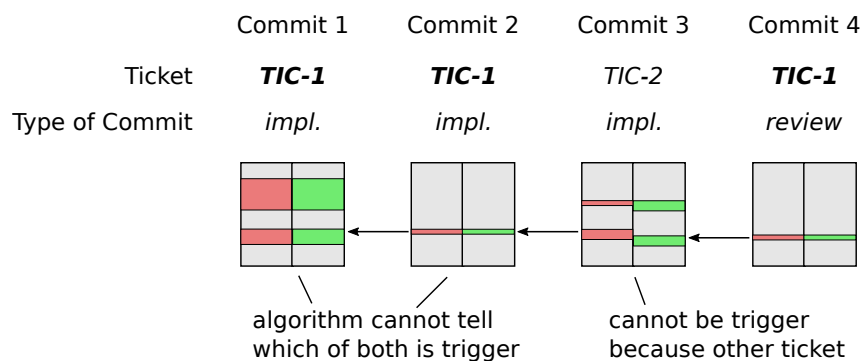
Figure 15.4: Example of a review remark (i.e., change in review commit) that is traced back in SCM history to find potential triggers

15.3.2 Intended Characteristics of the Model

With the approach outlined in the previous section, a mapping between change parts from the implementation commits and review remarks/changes can be created. The next step is to use this data to construct a classifier for change parts to be left out from reviews.

Various types of models are used in data mining techniques, e.g., rules, neural networks, regression models, etc. A suitable model for this study should meet the following requirements:

- The constructed model must be able to reach an adequate profit, as defined in Section 15.2.2.

- The model shall be discussed with the development team of the partner company, both to achieve better results and to increase user acceptance [365]. Therefore, it needs to be understandable by human developers.

- For each review, it shall be transparent why certain parts of the change are left out from the review, i.e., its decisions shall be *explainable* [88]. Furthermore, the reviewer shall be able to override this decision for certain parts.

To cross-check the requirements, I performed a survey among all available software developers of the partner company. The survey was handed out on paper and collected anonymously in the following days. The questionnaire asked for a rating of the importance of several requirements for the model and the modeling process on a 7-point Likert-type scale. It furthermore asked for a rating of several granularities to group code changes in the review into review remarks. For both questions, explanations and other free-text comments could be given. I received 13 responses, of which 3 answered that they are not experienced enough with the topic to reply to the remaining questions. The analysis is based on the remaining 10 responses.

Table 15.3 shows the aggregated results for the requirements' importance. The most important requirements are a low number of misclassifications and a quick review start. Here, "quick review start" is more a restriction on the usable features of the data, as all common model types are quick to evaluate. Next, and also with high importance on average, come the three requirements on the understandability of the model. The requirements with the least relative importance are the quick and effortless creation of the model and maximization of the number of change parts that the model classifies as 'no review needed'. When looking at the latter result in detail, there is a large spread in the answers, with some developers considering this as very important and others of the opposite opinion. One of the developers who considered a high number of 'no review' classifications as unimportant explained his answer: *"I consider*

Table 15.3: Survey results for the importance of various requirements for the prediction model and mining process. All ratings are on a scale from 1 (not important at all) over 4 (neutral) to 7 (extremely important). The requirements are translations of the German originals. Rows are ordered by mean rating.

Requirement	Rating			
	Mean	Median	Min.	Max.
The model classifies as few change parts as possible as "no review needed" by error (i.e., the leaving out leads to as few oversights as possible)	5.9	7.0	3	7
The model can be evaluated quickly at review start	5.5	5.5	3	7
I can see why a certain change part was classified as "no review needed"	5.3	6.0	1	7
The parts of the model have been checked by the development team before deployment to production	5.2	5.0	4	7
I can override/disable parts of the model so that the respective change parts remain in the review scope	5.2	5.0	2	7
The initial creation of the model from the raw data requires little human interaction	4.7	5.0	1	6
The model classifies as many change parts as possible as "no review needed"	4.4	4.5	1	7
The initial creation of the model from the raw data is quick	4.0	4.0	1	7

leaving out change parts in reviews as dangerous because it can give a false sense of security. Personally, I prefer thorough code reviews."

For the second question in the survey, the right granularity for counting missed review remarks, the developers preferred counting change parts, followed by counting lines. Counting at a coarser granularity, i.e., files or whole tickets, was considered inadequate by the majority of respondents.

> **RQ**$_{15.2.1}$: *The three most important requirements from the team are that the model provokes as few missed review remarks as possible, does not lead to additional waiting time in the review, and is transparent to and checked by the team.*

Concerning this section's goal to select an adequate type of model, the survey results support the initially stated requirements. A rule-based model seems well-suited to satisfy these requirements: Rules are a well-known concept for software developers, and a rule-based model is relatively easy to analyze manually. The classification task is binary, classifying each change part as either 'needs no review' or 'review'. Based on the intuition that it is possible to find some rules for change parts that definitely need to be reviewed, other parts that definitely need no review and a harder to classify rest that will conservatively be kept for review, rules should have the following form:

```
skip when one of
   (... and ... and ...)
   or (...)
```

```
    ...
unless one of
    (... and ... and ...)
    or (...)
    ...
```

Or, put mathematically: $skip := \bigvee_{r \in incl} \left(\bigwedge_{c \in r} c \right) \wedge \neg \bigvee_{r \in excl} \left(\bigwedge_{c \in r} c \right)$. This notion allows certain rules to be written more concisely than a simple disjunctive normal form but is still simple and easily explainable. Furthermore, every single rule in the disjunctions can be treated as a separate nugget of knowledge and can be used to explain and override the decisions of the review tool. The rules are restricted to propositional logic for the single conditions: 'less or equal' and 'greater or equal' for numeric features and 'equals' and 'not equal' for categorical features.

15.3.3 Mining Rules from the Extracted Data

There are two complications when using standard data mining algorithms on the extracted data: (1) The standard algorithms optimize for accuracy, not for profit. (2) They cannot take into account that remarks are not just simple classification labels but are objects with a distinguishable identity: A mining algorithm can exploit the identity of the remarks because it suffices to cover only one of the potential triggers for each remark. As an example, consider two potential triggers, one an addition of complex code and the other a minor change. Without taking into account that only one of these triggers is needed, a rule of the form 'minor changes need no review' would be unnecessarily seen as inaccurate. Section 15.4 shows that the generic rule mining algorithms RIPPER [78] and C4.5-Rules [404] indeed create inferior results in this study's context.

One of the variants of profit, as introduced in Section 15.2.2, is the primary target metric. But simply maximizing it on the training set can lead to overfitting. Simpler rules and rules that have fewer features can work better on unseen data, even though they seem worse on the training set. This thesis assumes that this problem is best dealt with by letting the development team select the final ruleset from a number of candidates with different characteristics. Simpler rules have the additional benefit of being easier to explain, and rules with fewer features lead to less implementation effort in a tool. Therefore, the mining task is treated as a multi-objective problem: The basic objectives are 'maximize saved effort', 'minimize the number of missed review remarks', and 'minimize rule complexity'. Saved effort and missed review remarks are used in favor of profit because the latter can be derived from the former and do not need an estimate of the cost of missed remarks. More details on the finally used objective vector are given in Section 15.4.1.

To be able to take the above problems and the multi-objective nature of the problem into account, we (Baum, Herbold) developed the GIMO approach for interactive, multi-objective rule mining. Its main characteristics, distinguishing it from other data mining approaches, are:

Multi-objective Instead of optimizing for a single objective, the approach considers multiple competing objectives and determines a Pareto front of solutions (see Figure 15.5). In this way, the user can decide a posteriori, with knowledge of the found solutions, which compromise of the objectives to settle for.

Geared towards domain expert feedback Many data mining approaches treat human feedback as an after-thought, if at all. In contrast, GIMO is designed to allow efficient collaboration of domain expert and computer. To reach this goal, it uses rules as a human-understandable model, allows the users to give interactive feedback in domain terms (e.g.,
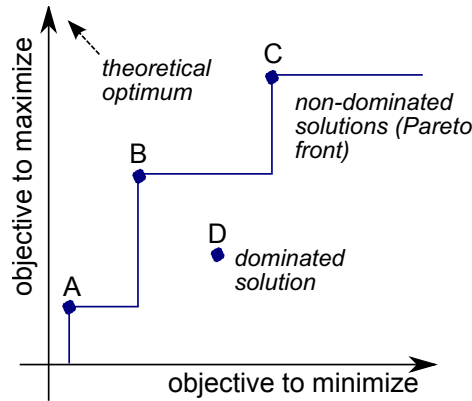
Figure 15.5: Example to illustrate the concept of Pareto-optimality. Solution B dominates solution D because it is better in both objectives. The other solutions do not dominate each other because they are better for one but worse for the other objective. They form the Pareto front.

as patterns of rules that do not make sense in the domain) instead of opaque tuning parameters, and integrates data mining and interactive data exploration and preprocessing. In contrast to many data mining algorithms that cannot be influenced once started, GIMO allows iterative user interaction at any time.

Domain-specific evaluation As shown above, evaluating the quality of a rule is not a simple matter of counting misclassifications of change parts. Instead, multiple change parts can act as triggers for the same review remark, and the approach assumes that it suffices to read one of these change parts to find the remark. Contrary to usual data mining algorithms, GIMO allows the integration of this 'set cover' aspect for model evaluation.

To keep the current chapter focused, it does not contain the details of the GIMO rule mining system. A detailed description is available in a separate report [30].

15.3.4   Feature Selection

A data mining model can only deliver good results if the used features fit the task. To select the features for the current study, I use two sources: (1) Features that showed good results in defect prediction studies, based on a survey of the literature. (2) Iterative analysis and idea generation ("open coding") based on the code change data. By combining these sources, the similarity between defect prediction and review remark prediction is taken into account, as are the likely differences and more specialized features. The selection is restricted to a subset of the potentially applicable features to limit the effort for this part of the study. Details on the used features can be found in Appendix F.

15.4   Application of the Approach within the Partner Company

After having outlined the approach in the previous sections, this section describes the results of applying it in the partner company.

15.4.1 Iterative Improvement of the Approach

The interactive rule mining system outlined in Section 15.3.3 was used to infer rules from the data. Input from domain experts on intermediate results was given by the thesis' author and by three other developers from the partner company. The field notes and interaction logs can be found in the online material [32]. The feedback can be categorized into specific feedback on the mining results and meta-feedback on the approach. This section describes the meta-feedback and the resulting improvements to the approach.

The other developers found the system helpful and considered it interesting to analyze the data with it. Apart from minor technical problems with the rule mining system, the domain experts identified two major problems:

Too much focus on a small fraction of large tickets. Initially, the algorithm focused too narrowly on parts of the data with large-scale changes, e.g., single tickets that changed many files. These are correct findings that shouldn't simply be removed as 'outliers', but they are not very interesting because they account only for a small fraction of the tickets and the review effort.

Too much noise in the data. Noise appears in the data in several ways: Identified remarks might not be remarks at all, or they might be traced to wrong triggers. These problems can be due to remarks that are follow-ups of other remarks (e.g., rename refactorings or overrides) or remarks for code not belonging to the ticket. Another form of noise is caused by not taking the severity of the remarks into account, e.g., if a whitespace correction influences the algorithm as much as a critical defect. I analyzed a random sample of 100 remarks from an early version of the data to assess the problem. When checking for quality of the remarks, 19% were no true remarks. When checking the tracing, the tracing was OK for 71% of the remarks. Of the remaining 29%, 9% were definitely wrong.

Based on this feedback, the data extraction and mining objectives have been adjusted to mitigate the problems:

Additional objectives. The approach initially started with four objectives: missed remark count (to minimize), saved record count (to maximize), rule complexity (to minimize) and number of used features (to minimize). These are generic objectives that could be applied to almost every data mining problem[2]. Using only these objectives gives rise to the above-mentioned "too much focus" problem. This problem is countered with three further objectives: (1) "Saved lines of code in Java files" captures the intuition that the main effort of a review is mostly spent on the source code. (2) "Log-transformed missed remarks" is calculated by counting each missed remark with the logarithm of the total number of remarks in its ticket (instead of 1 as for "missed remark count"). This objective puts less weight on large review changes, based on the intuition that these are often systematic. (3) "Trimmed mean of saved records per ticket" is calculated as the 20% trimmed mean [398] of the number of saved records per ticket. It puts less weight on rare tickets that have a large number of records. As a per ticket measure, it is also easy to interpret by domain experts. Table 15.4 shows the seven finally used objectives.

Additional target functions. The GIMO approach uses combinations of the objective values to guide the search. These are called 'target functions'. The additional objectives described above give rise to additional target functions, too. The most important of these is a family

---

[2]taking into account that "missed remark count" is similar to "false positives" and "saved record count" is similar to "true positives + false positives"

Table 15.4: The seven objectives used for the multi-objective data mining

Objective Group	Objective	Rationale
Minimize missed review remarks	Missed Remark Count	The fewer remarks are missed, the better. Simple metric.
	Log-transformed missed remarks	When there are many remarks in a ticket, it could be more likely that each single remark is systematic and, therefore, less important.
Maximize saved effort	Saved record count	The more change part records can be skipped, the better. Simple metric.
	Trimmed mean of saved records per ticket	By using the trimmed mean, outlier tickets with a large number of remarks have a lesser influence.
	Saved lines of code in Java files	Java LOC count could be a better approximation of review effort than the number of change part records.
Minimize rule complexity	Complexity (= Number of Conditions)	The fewer conditions a rule set has, the easier it is to interpret.
	Feature Count	The fewer features are used in a rule set, the easier it is to understand and the fewer implementation effort is needed in the review tool.

of per-ticket cost functions. With cost defined as negative profit as given in Equation 15.1 and the cost factor c as introduced in Section 15.2.2, it can be stated as:

$$\text{cost}_c = \frac{r}{t}c - s$$

In this formula, r is the log-transformed missed remark objective value, t is the number of analyzed tickets, and s is the trimmed mean of saved records per ticket.

Merging of remarks with identical content. One of the problems with determining the review remarks based on the changes in review commits is that systematic changes lead to a large number of 'remarks' whose number does not adequately represent the cost of missing one of them. In addition to the "log-transformed missed remarks" objective (see above), all change parts that represent the same textual change in the same commit are merged into one remark.

Removal of certain remark types. Another means to get rid of remarks with minor relevance is to delete certain types of remarks: Remarks with whitespace-only changes, remarks with changes to package declaration or import statements only, and other derived changes.

Manual cleaning of the data. I systematically checked tickets with a large number of changes or a large number of review commits and removed problematic ones, e.g., when there was a violation of the development process that led to unusable data.

15.4.2 Extracted Data

Before delving into the details of the mining, this section gives some characteristics of the extracted data. Table 15.5 shows statistics for the extracted training data. It can be seen that

Table 15.5: Number of commits, change part records and review remarks in total and per ticket for the extracted training data. Commits are subdivided into implementation and review commits. There are 6,005 tickets in total. All counts are after cleaning.

	Total	Min.	25% Qu.	Median	75% Qu.	Max.
				per Ticket		
Commits (impl. + review)	23,853	1	2	3	5	88
Commits (impl.)	15,365	1	1	1	3	65
Commits (review)	8,488	0	0	1	2	32
Change part records	703,706	1	10	27	77	53,594
Change part records (Java)	370,130	0	4	18	57	5,749
Review remarks	68,960	0	0	2	10	1,437

the distributions per ticket are all heavily skewed, with a few huge changes. Java is the primary implementation language, and therefore the majority of changes is performed in Java files. Next in frequency of occurrence are XML schema files. XML schema is heavily used for interface definitions in the company. Most other frequently occurring changes are due to test data and committed dependencies. The earliest analyzed commit in the training data is from March 2013, the most recent from July 2018. Of the 703,706 change part records, 439,626 have no association to a review remark at all. Of the remaining, a minority of 14,385 records is the only trigger for at least one review remark. These records must be triggers in the proposed approach, whereas the remaining records are only candidates for triggers.

15.4.3 Rule Mining Results

The results from the multi-objective interactive approach (called "MO_I" in the following) are compared to three other approaches: The results from the multi-objective algorithm, but without human interaction ("MO_A"), rule sets created by the RIPPER and C4.5-Rules rule mining algorithm [78][3], and a baseline of skipping a randomly sampled share of the records for each ticket. To make the information on review remarks and their potential triggers (see Section E.1) amenable for RIPPER and C4.5, the study took a conservative approach: Every record that is linked to at least one remark is labeled as 'must review' and records without a link to a remark are labeled as 'no trigger'.

There are no established guidelines on how to assess the mining results in a multi-objective, interactive setting. Relying on iterative human feedback makes cross-validation impossible, and many metrics (like precision, recall, or cost) are only defined for single rulesets and not for a Pareto set of rulesets. Therefore, several evaluation approaches are combined: (1) Checking where the results from RIPPER and C4.5 lie relative to the Pareto front of MO_I. (2) Selecting rulesets from the Pareto fronts for MO_I and MO_A and comparing the objective values of all four rulesets. (3) Asking the company's developers for their opinion on the rules. Section 15.4.4 presents the results of the developer assessment and the discussion with the development team. Section 15.4.5 shows the quantitative results for unseen test data. The results for the training data are given in Appendix G. For the discussion with the development team, the ruleset MO_I was further adapted. This adapted version will be referred to as 'SESSION'.

---

[3]the RIPPER and C4.5 baselines were mined by Herbold

```
skip when one of
  (changetype == 'DELETE')
  or (isNodeModules == 'true')
  or (packageAndImportOnly == 'true')
  or (whitespaceOnly == 'true')
  or (changeInHunkSize >= -0.5 and hunkCountInFile >= 147.5)
  or (filetype == 'jav')
  or (binary == 'true')
  or (fileCountInCommit >= 55.5 and hunkCountInCommit >= 2560.0)
  or (fileCountInCommit >= 274.0)
  or (fileCountInCommit >= 11.5 and srcdir == 'testdata')
  or (gitSimilarity >= 98.5)
  or (project == 'UnitTestRunner')
  or (project == 'TestPlugins')
  or (commitsSinceLastRemarkInFile >= 78.0)
  or (newLineCountInFile >= 12170.0)
  or (entropyCbMed <= 0.0919)
  or (visibilityChangeOnly == 'true')
```

Figure 15.6: Ruleset SESSION, i.e., the ruleset that was used in the discussion with the development team. It is based on MO_I.

All selected rulesets can be found in the online material [32]. To give an impression of the found rulesets, Figure 15.6 shows the ruleset SESSION. It is the least complex of the five rulesets, and similar to MO_I. Abstracting its specific contents a bit, it contains the following groups:

- Derived or too low-level changes, e.g., changes in imports or generated code[4],
- Likely systematic changes, e.g., very large commits[5] or additions in files with many changes,
- Changes that are low-risk due to previous checks (compiler, CI server, . . .), e.g., deletions or pure whitespace changes,
- Files that are empirically low-risk because there was no remark for a long time,
- Changes that are low-risk because they concern non-production code, e.g., the "UnitTestRunner" project, and
- Very non-surprising changes, i.e., changes with a low entropy compared to the rest of the code base.

The three rules that are responsible for most of the savings are `packageAndImportOnly == 'true'`, `whitespaceOnly == 'true'`, and `binary == 'true'`. When using only these three rules, the trimmed mean of saved hunks per ticket is 8.5 records/ticket. This is a share of 84% of the value for the whole SESSION ruleset. The most influential single rule is `packageAndImportOnly == 'true'`, with savings of 5.4 records/ticket.

RQ$_{15.2.2}$: *The interaction between mining tool and domain experts with the GIMO approach identified a collection of Pareto optimal rule sets. Of these, the domain experts selected the rule set shown in Figure 15.6 as the most promising one. On the training data, 84% of the savings of this ruleset are due to three simple syntactic rules.*

---

[4]The team uses the file extension "jav" to denote generated Java code.
[5]Even when a large commit does not contain systematic changes only, the chance of finding problems in it might be lower.

Table 15.6: Survey results for the subjective quality of the mined rulesets. All ratings are on a scale from -5 (extremely bad) over 0 (neutral) to 5 (extremely good). Rows are ordered by mean rating.

	Rating			
Ruleset	Mean	Median	Min.	Max.
MO_I	-0.21	0	-5	3
MO_A	-1.08	-1	-5	3
RIPPER_S	-3.25	-5	-5	0

15.4.4 Developers' Opinion on the Rules

The requirements survey (Section 15.3.2) revealed that the developers prefer to check the rules before using them. Therefore, the development team and the author discussed the ruleset SESSION in a joint session. Before that session, the team answered a survey for a subjective ranking for the four other rulesets. Each rule was printed on a separate sheet of paper, and the participants were asked to rate it on a scale from -5 (extremely bad) to 5 (extremely good). They were also asked to give reasons for their rating. The sheets were shuffled before handing them out and did not mention how the rules were obtained. A total of 14 developers filled out this survey and took part in the discussion.

Table 15.6 shows that the team members considered the ruleset MO_I to be the best, and RIPPER_S as worst. But even the best ruleset has a negative mean rating (-0.21). Analysis of the textual comments from the survey and the audio recording of the discussion sheds light on the reasons.

Especially the RIPPER results were criticized for being hard to understand and containing partial rules that looked nonsensical. Large size, use of negation and rules with many numerical thresholds were especially detrimental to understanding. Often, this led to worse ratings, but sometimes the rulesets were still rated neutrally based on trust that "there will be something good in there". Contrary to the other rulesets, two participants criticized MO_I for not filtering out further change parts.

For MO_A and MO_I, the criticism was more towards specific rules and features. Often rules were regarded as not explicit enough, i.e., they left a theoretical chance of defects slipping through. Additional conditions were suggested to reduce this chance. This problem is discussed further in Section 15.5. Very coarse rules that lead to the non-review of whole files or commits were also criticized. To the developers' intuition, these rules and the features used in them miss a strong link to the importance of the changes for review. Most criticized for missing this link were the time/day features. The latter was also disapproved for providing the opportunity to "game the system", e.g., when waiting for a specific day to avoid a review. Single feature rules, for example based on whitespace or entropy, were instead praised as intuitive.

There were two opposing points of view among the company's developers. Some agreed with the point of view taken in the current thesis: Leaving out parts during review is a cost-benefit-tradeoff, and the costs and benefits can be derived from empirical data. Others took on a point of view that opposed leaving out change parts in reviews. The latter group often argued with the theoretical possibility of defects, no matter how small the empirical risk. Instead of shrinking the review scope, the reviewers should take more time for reviews and make pauses to avoid overload. Looking again at the results of the survey on the requirement importance (Section 15.3.2), the vast spread of responses for leaving out many change parts can be explained

by these opposing views.

Based on the controversial discussion in the development team, the team decided to implement the skipping rules in its review process in a limited way for now: The classification and reasons for the classification of every change part shall be visualized, but they should not be left out entirely in an automated way. In this way, their consequences can be studied in practice.

RQ$_{15.2.3}$: *The ruleset obtained by the interactive multi-objective approach (MO_I) is rated best by the developers. The ruleset from RIPPER is rated worst. But even MO_I is rated only neutral on average. Likely reasons are a rejection of specific contained rules and a general opposition to the idea of leaving out change parts in reviews by some developers.*

15.4.5 Performance on Unseen Data

After creating and discussing the results based on the training data, I waited until mid-November 2018 and extracted the new data that had accumulated in these 3.5 months. This test set contains data from 311 tickets. After applying the same filtering as for the training data, the found rulesets were re-evaluated on the test data.

Figure 15.7 shows projections of the Pareto front obtained by the multi-objective algorithm with and without input from domain experts. It also shows the position of the four selected rules in the objective space. RIPPER and C4.5 are dominated by both Pareto fronts, and do not perform much better, and sometimes worse, than just skipping the review of random change parts. RIPPER differs from all other rulesets by having the form "skip all except . . . ". RIPPER and C4.5 are insensitive to the (high) cost of missed remarks and lead to rulesets with many missed review remarks. The RIPPER and C4.5 rulesets are also more complex than the MO rulesets. The exact numbers for these and the other objectives are shown in Table 15.7, and Table 15.8 reframes the results for savings and missed remarks as relative values.

Comparing the results to those on the training set (Appendix G) shows that MO_A's performance strongly degraded: The saved hunk count and the saved Java source line count plummeted, so that it does not filter out much at all on the test data. The trimmed mean of saved hunks is already low for this ruleset on the training data, indicating that the difference in performance might be due to focusing too much on large but rare events. A likely cause is that MO_A is the only ruleset whose creation did not employ an automated or interactive mechanism against this problem.

For the SESSION ruleset, twelve remarks are counted as missed. I analyzed these in detail: In six cases the remark would have been found anyway, i.e., they are false positives. For two of these false positives, the remark is real but not all triggers were identified, and the other four were follow-up changes and no real remarks. This leaves six cases in which a real remark would have been missed. Of these, two are negligible improvements. Another one is a build script maintainability issue that would have been missed because the respective commit was larger than 274 files. The most severe misses are three customer-facing documentation issues/typos that would have been missed because the file containing the documentation became longer than 12,170 lines. These misses echo the developers' criticism of coarse rules that miss a strong link to the importance of the changes for review (Section 15.4.4).

Looking at the objective values for the ruleset SESSION, it allows skipping of 25.2% of the records per ticket (trimmed mean) and of 23.2% of Java source code lines in reviews. According to the tracing mechanism, this skipping will lead to non-detection of 0.6% of the review remarks

153
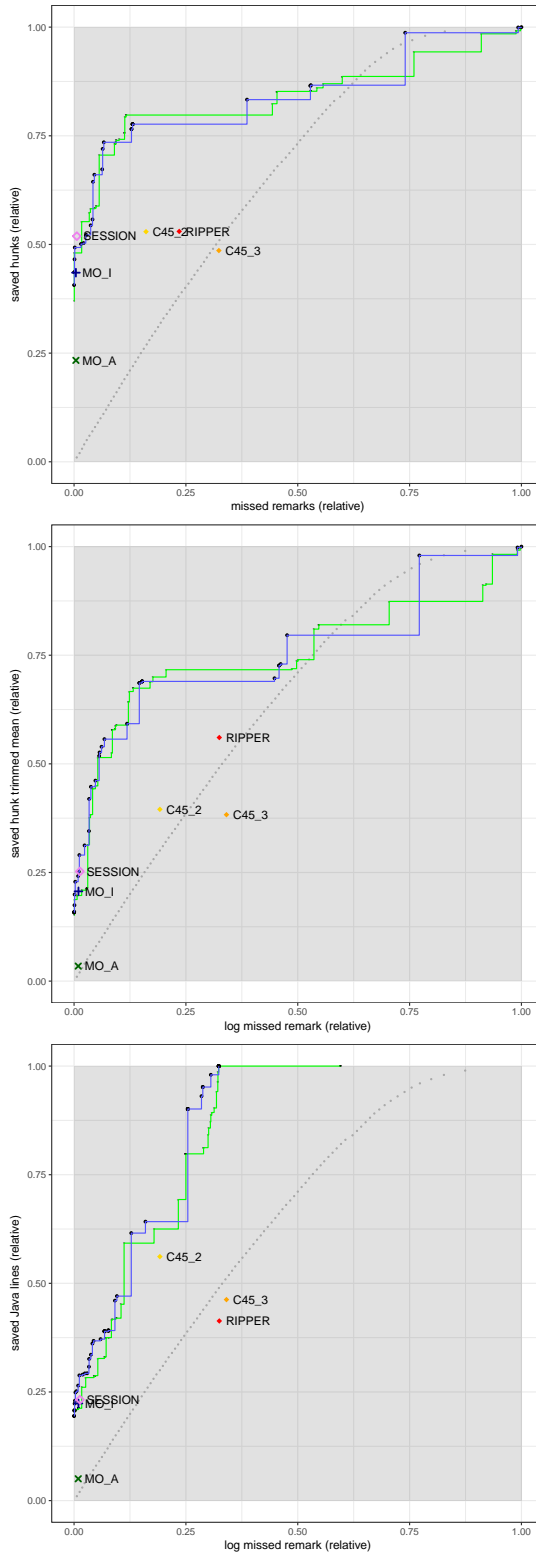
Figure 15.7: Pareto fronts and selected rulesets, evaluated on the unseen test data. The plots show two-dimensional projections from the seven-dimensional objective space. The gray dots show the baseline performance of leaving out a certain percentage of records per ticket; each dot corresponds to a percentage value, with results averaged over 100 random seeds.

154

Table 15.7: Objective values for the selected rulesets on the unseen test data

Ruleset	Objectives to Minimize				Objectives to Maximize		
	Compl-exity	Feature Count	Missed Remark Count	Log-Transf. Missed Remarks	Saved Record Count	Tr.M.[1] Saved Records per Ticket	Saved LOC in Java Files
SESSION	40	17	12	4.1	13,998	9.5	11,425
MO_I	58	17	8	3.1	11,733	7.8	11,004
MO_A	184	24	8	2.9	6,293	1.3	2,469
RIPPER_S[2]	200	17	539	110.1	14,166	19.2	18,784
RIPPER	342	25	456	100.9	14,294	21.2	20,338
C4.5_2	151,084	39	312	59.6	14,277	14.9	27,625
C4.5_3	63,872	38	628	105.8	13,106	14.5	22,756
Max. Valu[3]	∞	52	1,940	310.7	26,968	37.8	49,199

[1] Tr.M. := trimmed mean

[2] During the team session it was decided to remove one further ticket from the training data. RIPPER is the rule set learned with the final data, RIPPER_S is based on the older data and was used in the survey.

[3] The last row shows the total count / maximum possible value for the respective objective.

Table 15.8: Relative objective values (i.e., percentages of the maximum) for the selected rulesets on the unseen test data. Tr.M. := trimmed mean

Ruleset	Objectives to Minimize		Objectives to Maximize		
	Missed Remark Count	Log-Transf. Missed Remarks	Saved Record Count	Tr.M. Saved Records per Ticket	Saved LOC in Java Files
SESSION	0.6%	1.3%	51.9%	25.2%	23.2%
MO_I	0.4%	1.0%	43.5%	20.7%	22.4%
MO_A	0.4%	0.9%	23.3%	3.5%	5.0%
RIPPER_S	27.8%	35.4%	52.5%	50.8%	38.2%
RIPPER	23.5%	32.5%	53.0%	56.1%	41.3%
C4.5_2	16.1%	19.2%	52.9%	39.5%	56.1%
C4.5_3	32.4%	34.1%	48.6%	38.3%	46.3%

(resp. 1.3% when using the log-transformed value). Based on the raw objective values, approximations for profit can be calculated. As motivated in Section 15.2.2, these values depend on a cost factor that determines how much more costly missing a remark in a review is compared to reviewing a change part. A break-even value that determines when the profit will become positive can be calculated. When using the log-transformed remark count and the trimmed mean of saved records per ticket, this break-even value is 726, i.e., using the ruleset has a positive profit as long as missing a review remark is less than 726 times as costly as reviewing a record. When using Java lines of code as the approximation of effort instead, the profit is positive as long as missing a review remark is less than 2,793 times as costly as reviewing a line of Java code.

The three simple syntactic rules that are responsible for most of the savings for the SESSION ruleset on the training data are responsible for most of the savings on the test data, too. `packageAndImportOnly == 'true'` is most influential, `binary == 'true'` second and `whitespaceOnly == 'true'` third. When using only these three rules, the trimmed mean of saved hunks per ticket is 7.8 records/ticket. This is a share of 82% of the value for the whole SESSION ruleset.

> **RQ**$_{15.2.4}$: *The ruleset SESSION, and the closely related MO_I, both perform well on the unseen test data. They allow savings of more than one-fifth of reviewed records and Java LOC. In turn, they lead to about one percent of missed remarks. The RIPPER and C4.5 rulesets lead to much higher numbers of missed remarks. MO_A filters out very little on the test data.*

15.5 Discussion

The study could successfully extract data and mine useful rules, but some problems were not yet satisfyingly solved:

Noise. Section 15.4.1 describes a significant amount of noise found in the data, due to failed assumptions of the tracing algorithm. Other problems are caused by non-obedience to the development process, e.g., when remarks are not fixed in the proper ticket or the ticket state is not changed correctly. Noise is a significant problem: The most relevant target for the mining process is the profit of using the ruleset. The cost of missing a remark is magnitudes larger than that of reviewing a change part. Therefore, the profit metric behaves similarly to precision, which is known to be vulnerable to noise [264]. Many potentially interesting rules might be obscured by the noise in the data.

Unknown individual costs. The cost per individual missed remark is not known. This can lead to wrong incentives for the mining algorithm, which might opt for a rule that leads to the oversight of one critical error instead of another that misses two trivial issues. The same problem exists for the cost of reviewing change parts, but this cost can at least be approximated for example by the number of lines of code.

Reliance on Occam's razor. Ruleset complexity is one of the algorithm's objectives for two reasons: (1) Simpler rules are easier to understand. (2) There is the hope that of two competing and otherwise similar rules, the simpler one is the better model of reality (Occam's razor). Albeit useful, Occam's razor is only a heuristic that fails sometimes. An example that was criticized by some of the company's developers illustrates this: The algorithm selected a rule to ignore whitespace-only changes. This rule is straightforward, and it is also highly successful, as there are many whitespace-only changes in Java files that do not lead to review remarks. But in other file types, whitespace changes are much more dangerous and remark-prone. When changes in these files occur rarely, the 'ignore whitespace-only changes' rule will seem better than the 'ignore whitespace-only changes in Java files' rule, albeit the latter better represents the underlying domain.

Reliance on historical data. A general problem of every empirical approach for rule creation is that the rules are built for past data. Changes in the project structure, development processes, or other areas can make found rules obsolete. One example is the growth in file size that led to the missed remarks described in Section 15.4.5. Another example is the "isNodeModules" rule

in MO_I, MO_A, and SESSION, which relates to the practice of committing certain libraries to the SCM. This practice was stopped by the development team so that the rule will not lead to savings in the future. Besides input from domain experts, regular re-training can help to reduce this problem.

Amplification of reviewers' weaknesses and self-fulfilling prophecies. If a change part does not trigger review remarks, this can have two causes: There is no problem, or the reviewer did not find the problem. The latter case is disputed: When there are changes in which the reviewers consistently overlook problems, this weakness of the reviewers might be institutionalized as a skipping rule. This rule deprives the reviewers of the remaining small chance to find the issue. The problem could be reduced by combining a model for skipping with the SRK model discussed in Section 15.2 or with a general defect prediction model. A variant of the problem occurs when data from a team in which skipping rules are in use is used to re-evaluate the rules: As the reviewers did not see the change parts that are matched by the rules, they probably left few remarks for them. So, the existing rules will look good no matter how good they really are – a self-fulfilling prophecy.

Much ado about .. not much. A large share of the savings of the SESSION and MO_I rulesets are due to simple syntactic rules. This leaves the impression that similar benefits for the company could have been reaped in a more lightweight way, without going through the hassle of extracting the remark data and performing a large-scale data mining study. It is open whether less noisy data would allow the extraction of less obvious high-impact rules or whether there is just nothing to be found. Please note that the criticism of the high effort applies mostly to the action research view of the study; I still consider it worthwhile to gather and analyze empirical data in detail from a basic research perspective.

Most of the mentioned problems are not unique to the current study; they are shared by many other repository mining studies. This might contribute to their still relatively low use in industry.

15.6 Validity and Limitations

$RQ_{15.1}$ is only studied theoretically, deducing arguments from previous findings. To show that these arguments really hold, empirical validation is needed.

The answers to the remaining research questions ($RQ_{15.2.1}$ to $RQ_{15.2.4}$) provide a first step towards such empirical validation. Still, several limitations apply to them as well. One of the central limitations stems from being a single case study, which makes the generalizability of the results to other contexts hard to judge. This applies to the mined data and found rules as well as to the results of the surveys and discussion sessions. As an example, the high savings due to leaving out changes in import statements are probably influenced by the use of version numbers in some package names and by a positive attitude towards refactorings, both of which might not exist in other contexts.

Another limitation is that the study does not measure the final outcomes of introducing the model in practice. A smaller review scope leads to better results (Chapter 13) and the case study results indicate that applying the model will safe effort, but it does not measure the construct 'effort' explicitly. The validity of many other of the measured constructs depends on the correctness of the mining and extraction programs. Many parts of the extraction and mining source code are tested with automated unit tests. But as a heuristic approach, the mechanism to count the number of missed review remarks can still fail. It might do so in two ways: identifying

too many or too few missed remarks. By manually checking a sample of the remarks, the 'too many' case could be assessed. But the 'too few' case, e.g., review remarks that were fixed in a commit that was not correctly labeled with the ticket number, is only assessed qualitatively.

A usual practice in data mining studies is to use repeated cross-validation to statistically judge the quality of the results. This is not possible with an interactive approach. Instead, the study uses a training set and a separate test set. Having only one test set leads to less reliable results. On the other hand, using a test set collected after mining is a very realistic way of performance evaluation. To put the interactive, multi-objective mining approach in context, it is compared to RIPPER and C4.5 as a baseline. In defect prediction studies, other mining approaches outperform RIPPER and C4.5 [366]. But these approaches, like random forests, lead to opaque models that do not meet the team's requirements. Therefore, a rule mining approach is an adequate baseline.

Various threats apply to the opinions gathered from the developers. In the interactive evaluation sessions, the participants needed more time than initially expected to understand the system, as they were overwhelmed by the current UI of the system. This limited the amount of feedback that could be gathered in the available time. The opinions on the mined rulesets were gathered directly before the discussion session. As some of the presented rules were clearly suboptimal, this might have put some developers in a negative mood for the discussion. Another threat here is reactivity, in particular, because the researcher presenting the approach and the rules was also a colleague.

For the analysis of the qualitative data, best practices from qualitative data analysis, like transcribing the audio recordings and performing several passes of open coding, were used. Still, having only one coder amplifies the risk of researcher bias. To allow others to judge this and other potential biases, the session transcripts are available [32]. All other raw data is also made available, albeit the dataset was anonymized to protect confidential company data. With an interactive data mining approach, it is not sufficient to make the data mining algorithms and scripts available to allow others to double-check or recreate the results. Therefore, logs of the interactive sessions are also provided as an audit trail.

15.7   Related Work

Besides the discussion of cognitive-support review tools in the current thesis, positive effects of reducing the review size are also discussed in other works (e.g., [315, 318, 319]). Section 15.2 also relates to another theory from cognitive psychology, the SRK taxonomy that divides human cognitive processing into three modes that differ in the degree of automation and efficiency of processing [311]. A general analysis of the utility of these and other theories for software engineering research is performed by Walenstein [393].

SZZ [353] is the standard algorithm for tracing defect fixes to the introducing changes in defect prediction studies. Kim et al. [199] proposed a variant of this algorithm that reduces the noise in the extracted data. Like the proposed tracing algorithm for review remarks, they skip changes that cannot be a cause/trigger. The problem of noise in defect prediction is studied by Kim et al. [198].

Review remark prediction is not the same as defect prediction: A change part might contain a defect that does not lead to a review remark, and the different assumptions during tracing also lead to differences in the raw data (see Section E.5). Still, review remark prediction and defect prediction are related. Defect prediction and repository mining are vast research areas,

so this section can just give a small overview of the literature.

This chapter's approach classifies at the level of change parts. Most defect prediction studies work at a coarser granularity, e.g., predicting defect proneness for whole changes (also called "just-in-time" prediction) or for methods, files or components. An early study to predict the risk of a software change was performed by Mockus and Weiss [272] at Bell Labs. Shihab et al. [340] use input from industrial developers to assess the risk of changes. One of the problems of the current study is that it considers the severity of the review remarks only to a limited degree. Shihab et al. [341] tackle a similar problem for just-in-time defect prediction and study "high impact" fixes. Results of just-in-time prediction at Google were disappointing [233], but Tan et al.'s case study at Cisco [365] and especially Nayrolles and Hamou-Lhadj's case study at Ubisoft [280] show that an elaborated approach can bring just-in-time defect prediction to industrial usefulness. There are further defect prediction studies with change granularity (e.g., [191, 197, 345, 355]). Ray et al. [314] show that the entropy of code is related to its defect density and go down to the level of single lines. Results of Pascarella et al. [293], when re-evaluating a study by Giger et al. [139], indicate that the current state of the art of method-level defect prediction does not lead to results that are satisfying in practice. Shippey [344] analyzes the relation of AST patterns to defective and non-defective methods. He finds associations for defective methods, but not for non-defective methods.

The current study argues that the meta-parameters of many data mining techniques are hard to interpret for domain experts. Other researchers (e.g., [366]) use automatic approaches to optimize these meta-parameters. Arisholm et al. [14] criticize that often used evaluation measures like precision and recall do not directly relate to cost-effectiveness. As Kamei et al. [190] show, effort-aware evaluation of models leads to different conclusions. A benefit of the proposed approach is that it is easy to include domain-specific evaluation measures. The decision to go for a multi-objective mining technique was encouraged by Fu et al.'s promising results with DART [133], which can be regarded as an ensemble of rules created with a multi-objective approach. Further studies on defect prediction are surveyed by Hall et al. [153], Radjenović et al. [309], and Malhotra [246].

Review remarks can relate not only to defects but also to other quality problems. These, too, can be found with data mining, for example like Fontana et al. [126] who predict code smells.

This thesis focuses on ways to support the reviewer. Another avenue to improve code reviews is to create automated review agents [69], i.e., programs that directly create review remarks. Just-in-time defect prediction is one possibility to create such automated reviewers [121, 280, 326]. Another possibility is to use results from static analysis [46, 63, 119, 170]. Automated reviewers can already work during check-in [28, 369] or even in parallel to development [244]. Section 15.2 discusses sorting change parts by their importance for review. Similar ideas have been studied by other researchers, e.g., by Lumpe et al. [236]. The idea to focus reviewing on "sections where finding defects is really worthwhile" can already be found in Gilb and Graham [140, p. 74], albeit they relate more to the severity of defects and not to the probability of finding them. Begel and Vrzakova [45] study eye movements of reviewers and find that developers focus on relevant portions instead of the entire text, which supports this chapter's idea of automatically classifying parts as irrelevant. Huang et al. [175] propose a machine learning approach to identify the 'salient class' in a change, i.e., the class whose change is at the core of the commit. This could be a further criterion to rate the importance of code changes for review. The idea to leave out lines of lesser importance to improve understanding has been studied in the context of end-user programming by Athreya and Scaffidi [16].

The proposed approach is bottom-up and empirically determines change parts that are ir-

relevant for review. Other researchers instead start by assuming that certain types of changes are irrelevant. Refactorings are one such type, and Ge et al. [136] propose to use refactoring detection in code reviews. Thangthumachit et al. [371] use refactoring detection to improve the understandability of source code changes. Going beyond refactorings, Kim et al. [196] describe how to discover systematic code changes and Zhang et al. [413] propose a tool to interactively match systematic changes in code reviews. Simple syntactic rules (e.g., whitespace and import statements) are responsible for a large part of the savings with the selected rulesets. Similar results could be obtained by using semantic or otherwise improved diff algorithms [13, 125, 192, 410].

The current chapter studies the importance of change parts for review. Other aspects of review data have also been assessed in data mining studies: For example, Gerede and Mazan [138] predict at the coarse granularity of whole patches whether they will lead to review remarks. Padberg et al. [290] predict the defect content after reviews and Kononenko et al. [206] study factors that influence review quality in Firefox subprojects. Bird et al. [55] have deployed a code review analytics platform at Microsoft, which could also be useful to determine irrelevant change parts. It was used by Bosu et al. [61] to develop a model to distinguish useful review remarks from less useful ones (i.e., noise).

The current study uses a multi-objective rule mining meta-heuristic and domain feedback to create comprehensible rules. With the same goal, Vandecruys et al. [386] use the AntMiner+ meta-heuristic for mining. Other researchers use a two-step process: First, a black box model is created, and as a second step this model is transformed into a comprehensible model [273]. This approach is not without criticism [327]. Still another approach is to create explanations for the model's decision upon request [88, 365]. Multi-objective meta-heuristics were also used in other software engineering studies, often based on genetic algorithms. They are used for the model creation itself [73, 92], but also for feature selection [72] or model refinement [68, 330, 406]. There are also studies that explicitly incorporate iterative human feedback, e.g., to determine patterns for implicit coding rules [254], common defects [400] and requirements tracing links [166].

$$\diamondsuit$$

Summing up, this chapter discusses how classification of change parts can support the reviewer. It shows how a bottom-up repository mining approach can be used to build such a classifier. The approach contains a novel extraction and tracing algorithm for review remarks and makes use of a multi-objective, interactive data mining system. Albeit there are some problems with the approach, especially with noise in the data, it leads to a substantial reduction in review scope. Consequently, the most promising found rules have been implemented in CoRT, with positive user feedback.

Part IV

Conclusion

16

Conclusion

This final chapter summarizes the thesis, discusses a selection of its implications and hints at major areas of further research related to code reviews.

16.1 Summary

This thesis started out with the idea to improve computer support for code reviews in practice that goes beyond the book-keeping and data-handling support of current tools. In its three main parts, the thesis gives an extensive discussion of the state of the art and the practice (Part I), shows how an instrumented review tool is used as a foundation for code review research in a partner company (Part II), derives the concept of cognitive-support code review tools, and provides extensive empirical studies on several ways to provide such cognitive support (Part III).

16.1.1 Cognitive-Support Code Review Tools

The systematically derived notion of "cognitive-support code review tools" forms the central contribution of this thesis. Such tools extend the current state of the art of review tools with features that reduce the cognitive load of the reviewer during checking. An empirically and theoretically founded catalog of essential requirements for cognitive-support code review tools, including general core features of review tools, can be found in Appendix A. This catalog summarizes the main findings on review tools from this thesis in a way that is geared towards tool developers.

Chapter 12 presents a number of ways to provide cognitive support. Two of them are studied in detail: Ordering of change parts (Chapter 14), and classification of change parts by importance based on repository mining (Chapter 15). Both approaches are derived and tested empirically, and both have shown their practical viability with an implementation in the review tool CoRT. A limitation of this thesis is that the controlled experiment to test the proposed ordering could not show a statistically significant improvement. There are some indications that this could be a problem of statistical power. Regarding the study on change part classification in Chapter 15, reducing the review scope by more than 20% while missing out very few remarks is an opportunity that should not be left untapped. But when it comes to how to establish

the model underlying this mechanism, repository mining does not seem to be the best way at the moment. Most of the found savings are due to simple syntactic rules, and it seems that most of these could have been conceived without laborious repository mining, albeit with worse empirical underpinning.

In addition to the analysis of cognitive-support, improvements of the scientific foundations of code review tools in general are a contribution of this thesis, too. Based on the analysis in Part I, Part II describes the CoRT code review tool, which is used both as a practically-used review tool in a partner company and as a platform for code review research. The thesis presents two studies that influenced the development of CoRT. In the first study (Chapter 10), the effects and contextual factors that influence whether pre-commit or post-commit reviews lead to better results regarding efficiency, quality, or cycle time are analyzed. It shows no striking differences in most cases, with a disadvantage of pre-commit reviews in terms of cycle time usually being the main difference. The study's more detailed results are condensed into heuristic rules for the use in practice. The second study (Chapter 11) compares different ways to present source code diffs. Albeit no statistically significant performance difference could be measured, it shows a preference of developers towards colored and aligned presentation.

16.1.2 Results on Code Reviews beyond Tools

State of the practice. Part I of this thesis contains an extensive discussion of the state of the art and state of the practice regarding code reviews. Particularly, both the interview study and the survey on code review use in practice are the largest scientific studies of the respective type done to date. They confirm the observation from other researchers [18, 318] that code review in practice is converging towards a change-based process, which is used by development teams to reach a combination of goals. But the studies also show that there is a lot of variation in the details of these processes, and the thesis proposes a classification scheme (see Appendix B) to capture these variations. Large parts of the survey's questions are based on this classification scheme, and they are made available for re-use by other researchers.

Hypotheses on the use of reviews. Following the Grounded Theory methodology, the interview study led to several hypotheses on the use of reviews, of which some could be confirmed in the survey (Chapters 5 and 6). Particularly, the thesis provides evidence that the use or non-use of code reviews depends to a large degree on cultural factors. Furthermore, there is evidence that code review is most likely to remain in use if it is embedded into the development process (and its supporting tools) so that it does not require a conscious decision to do a review.

Systematic overview of review tools and reading techniques. As a final contribution of Part I, Chapter 7 contains a semi-systematic literature review on code review tools and contains an analysis of the common ideas behind several code reading techniques.

Influence of working memory capacity and change size on review performance. Part III's main purpose is to present the rationale and formation of cognitive-support review tools, but it also contains findings that transcend this purpose. The experiment in Chapter 13 provides evidence that the reviewer's working memory capacity is associated with code review effectiveness for certain kinds of defects. Additionally, the thesis provides a confirmation from a controlled experiment that review effectiveness is higher for smaller code changes, a fact that was widely assumed but not studied systematically so far.

16.1.3 Methodological Advances

A wide variety of research methods are used in this thesis, and novel and significantly improved methods form a contribution of this thesis on their own. Specifically, software engineering research could benefit from the following methods and practices:

Heuristics derived from simulation. Software process simulation has been proposed as a means to support decision making in development teams. But software process simulation is expensive [8], too expensive for small companies or agile teams. It is better thought of as a means for theory development [91]. Heuristic rules are an efficient way of communicating theories to developers, and Chapter 10 uses a combination of simulation and rule mining to develop such heuristics. Other researchers could use a similar approach for research questions that are not easily amenable to direct empirical evaluation.

Machine learning with the human in-the-loop. A basic theme underlying this thesis is that the best results can often be reached by combining the strengths of the human and the computer, instead of focusing on only one of them. Still, much research on software repository mining neglects the human side. The approach taken in Chapter 15 embraces iterative human feedback. It should encourage other authors to try multi-objective and interactive approaches to data mining when domain feedback and acceptance are essential.

Combination of systematic theory generation and theory testing. Iteration between theory generation and theory testing is a basic principle of science, and this thesis used it in several of its studies with good results.[1] Software engineering as a discipline could benefit from embracing it more thoroughly.

Open science. The datasets and algorithm implementations for the studies of this thesis were made publicly available as far as possible. The thesis also benefitted from publicly available implementations and results of others. Software engineering research is slowly moving towards open science, and perhaps this thesis can encourage others to follow and increase the pace.

16.2   Implications of the Findings

This section presents selected implications of the findings that go beyond cognitive-support code review tools. The detailed implications for building code review tools are summarized as 'essential requirements' and 'realizations' in Appendix A.

Research on improvements to change-based code review can have large practical relevance. The studies in Part I of this thesis confirm Rigby and Bird's [318] observation of convergence towards change-based code reviews in practice. Comparing the survey results to those of Ciolkowski, Laitenberger, and Biffl [75], the raw survey numbers indicate a large increase in the use of code reviews in the last 15 years. Another survey by Winter, Vosseberg, and Spillner [403] also shows an increase in the use of reviews. This implies that research on improvements to code reviews can have a large practical relevance.

Software Engineering research should not focus on open source development and large companies only. Many of the variations between the processes described by Rigby and Bird [318] could be observed in Part I's interview and survey data, too. However, this thesis couldn't observe as much convergence as they do regarding the subtleties of the processes. In part, this is probably due to a more fine-grained comparison of the processes, but there are

---

[1]Like the next point, this point is neither new nor unique to this thesis, but subjectively under-used in current software engineering research practice.

contradictions, too, for example for pre-commit vs post-commit reviews. Assuming that most of these differences are rooted in different contextual characteristics of the respective study samples, this is an indicator that results from studies of open source software development can be valuable for commercial development, but it is also a warning that results from large companies and open source development are not fully generalizable to smaller companies.

Proposed Software Engineering techniques need to fit the real-world context. By strengthening the evidence that using rules or conventions to trigger code reviews helps to keep code review use from fading away, Chapter 5 provides a partial explanation for the observed dominance of change-based reviews. As a more abstract consequence for future software engineering research, this finding strengthens the case for software engineering techniques that not only work in isolation but are also able to survive in the environment of a software development team. The low number of teams using perspective-based reading or a similar technique for code review could be an example for such a mismatch: There is little use in perspectives when there is only one reviewer.

Review tools should contain cognitive-support features. To have an impact on practice, improving a review tool is more effective than publishing a research paper, at least according to the findings from Part I's interviews. As argued in Chapter 12, better cognitive support is a promising opportunity for improvement, and Part III shows the viability of several such options. They all share the benefit that once such tooling is available, the effort of using it in a team is very small, compared to classic reading techniques that have to be taught to every reviewer. The observed variations in review processes and contextual factors imply that it is hard for a single tool to fit all practical contexts. This is one of the reasons to summarize the results abstractly in Appendix A so they can be used by other tool-builders.

Reviewers' mental resources matter. Chapter 13 shows that the reviewer's working memory capacity is associated with code review effectiveness for certain kinds of defects. Given that working memory capacity can be measured computerized in around 10 minutes per participant, it is reasonable to recommend researchers to measure it as a potential confounding factor in future studies. This finding also has practical implications for reviewer recommendation [376]: Code changes with more potential for delocalized defects could be assigned to reviewers with higher working memory capacity; moreover, working memory capacity could be used when distinguishing between reviewers for critical and less critical code changes in an attempt to find a globally optimal reviewer assignment.

Defect types matter. The finding that the factors that influence detection effectiveness differ between defect types leads to several new questions: Which further factors influence effectiveness for other defect types? And which other defect types or defect properties are relevant at all? Future studies should investigate better review support for defect types that are currently not found easily.

The problem of noise in review remark prediction needs to be solved. The study of change part classification in Chapter 15 establishes that review remark prediction is a distinct application area in repository mining and that there is much open work in this area. This thesis only follows one of the possibilities outlined in Section 15.2.1, and further delving into the distinction between knowledge-based and rule-based cognitive processing in reviews could lead to interesting results. It seems advisable to first tackle the open problems of the chosen approach, most importantly the reduction of noise in the data.

16.3   Next Steps in Code Review Research

To conclude the thesis, this section looks at promising directions for future code review research.

Further computer support for reviews. The first idea is also the one most related to this thesis: Chapter 7 analyzes the principles underlying various code reading techniques, and some of these principles are integrated into the developed tool, in addition to the idea of cognitive support. Further research could try to combine more of these principles with computer support. Additional support could also be gained by asking the author to provide further information for the reviewer and the tool [87]. This thesis tried to lessen the burden on the author and not mandatorily demand such additional work, but giving the option to add, e.g., information about importance could augment the approach.

Instrumentation of other review tools. By deploying a code review tool into practice that was explicitly designed as a research platform, this thesis could gather rich empirical data. Many other code review studies rely on data from more widely deployed review tools, with the benefit of a large empirical basis but the drawback of lacking information for several possible research questions. Researchers could try to collaborate with review tool builders to integrate further lightweight data collection facilities and extension points into popular review tools. This investment could pay off in a few years in a much better empirical basis for code review research.

More research on cognitive processes in reviews. The third research idea is based on the findings of Chapter 13: It is by now well-established that code review effectiveness is lower for larger changes. But the cause for this effect is not known. Several hypotheses have been proposed by this thesis and others: (1) Large changes lead to mental overload, (2) large changes lead to lower motivation, (3) large changes seduce the reviewer to review faster than optimal, (4) the mental focus of the reviewer fades after some time, and larger changes need more time, (5) larger changes increase the risk that delocalized defects are spread out, which in turn reduces the chance of finding them. Basic research on the cognitive processes during code review could shed light on the combination of these hypothesized effects that leads to the observed decrease in effectiveness. This topic touches on the more general question of whether the model of mental and cognitive load is adequate to explain performance in code reviews. The results of the thesis generally support the model, but it could be too simple and abstract to be useful to guide research. For example, it needs improvement for the currently quite fuzzy definitions of mental and cognitive load. Another mechanism that would benefit from further quantitative evaluation is the association between code understanding and review performance. Research can be carried out to investigate an extended version of the model, more specific for code reviews, and test it more thoroughly.

Fun in reviews. In the interviews and experiments of this thesis, many developers mentioned that they have less fun reviewing code than developing it, whereas others seemed to like reviewing. This was also observed in previous research [180], but is somehow surprising: Code reviews are similar to 'spot-the-problem' puzzles, and many people do such puzzles for leisure. Gamification of code reviews has been studied to some degree [335], but future research could investigate reasons for disliking reviews and propose techniques to make them more fun. One idea is that, in contrast to puzzles, reviews are lacking positive feedback for the reviewer. Another, related to the mental load hypothesis of this thesis, is that reviews often overload the reviewers and put them out of their 'flow' zone [84].

Training for reviewers. The reviewer and his or her fit to the reviewing task significantly influence review performance (see Chapter 12). This thesis briefly discussed reviewer recom-

mendation as a way to exploit this fact, and also why reviewer recommendation is often of little practical use. But reviewer recommendation is not the only possibility to have better reviewers for the task – reviewer education is another. The data gathered for the experiment in Chapter 13 shows a correlation between reviewer experience and performance. It also hinted at a much better review performance for defects that can be found with rule-based cognitive processing [311]. Deliberate practice [115] could be used to train these skills, and computerized tools could support this training. As the mental assessment of possible solutions is also a central skill when developing code [392], such review training could even lead to better general programming performance.

Culture in development teams. A final result that hints at future work is the observed influence of team and company culture (Chapter 5). When considering the software engineering industry as a whole, the potential gains of faster wide-spread adoption of best practices dwarf the benefits of improvements to specific techniques. Research on the intersection between business administration and software engineering can study how development team culture can be improved.

$$\diamondsuit$$

The main text of this thesis is now about to end. It started with an assessment of the state of the practice, based on several empirical studies and the literature. Motivated by the lack of a good code review tool that fits the context of the partner company and can be used as a platform for code review research, the second part of this thesis then described the code review tool 'CoRT' and two studies that supported its design. Finally, the notion of cognitive-support code review tools was proposed, tested, and two possibilities for cognitive support were studied in detail: Ordering of change parts, and classification of change parts as irrelevant for review. My hope is that several of the proposed ideas will make their way into other code review tools so that they can benefit a wider variety of software development teams. And I hope that some of the results of the thesis and some of the ideas for future work proposed in this final chapter will spark the interest in code reviews in other researchers. Their improvements to the understanding of the requirements of practice and of the cognitive processes during reviews could then lead to still better review support.

Part V

Appendix

A

Essential Requirements for Code Review Tools and Possible Realizations

This thesis started out with the goal to show how better tool support can help to improve code review in industry, and it shows and evaluates several possibilities. The findings are summarized under the label 'cognitive-support code review tools', and many of them are implemented in the tool CoRT. But one of the results in the first part of this thesis (mainly Chapter 6) is that there is a lot of variation between the review processes of teams in industry. One tool will not fit all. Other tool-builders need to integrate the findings into other code review tools. To ease the task of these tool-builders and to summarize the respective findings, this chapter condenses them into essential requirements for building code review tools, and lists realizations from this thesis for these requirements. Similar to patterns [6], these shall form a toolbox that other tool-creators can use. Figure A.1 shows an overview of the essential requirements that are discussed in the following.

After each 'realization', small icons indicate the strength of the respective evidence. Their meanings are:

Strong quantitative evidence from this thesis. A controlled experiment showed conclusively that the realization leads to the intended effect.

Weak quantitative evidence from this thesis, e.g., correlational results from a survey or promising but inconclusive results from an experiment.

Strong 'action research' evidence from this thesis. The realization is implemented in CoRT and was explicitly mentioned as important/beneficial by the users, or it is a standard feature of most review tools that are in wide-spread use.

Weak 'action research' evidence from this thesis. The realization is implemented in CoRT and perhaps a few other tools and is known to be used.

Strong qualitative evidence from this thesis. The realization was mentioned as important/beneficial by various interviewees or survey participants.

Weak qualitative evidence from this thesis. The realization was mentioned as important/beneficial by a few interviewees or survey participants.

Strong deductive evidence. It can be clearly derived from the literature, or is perhaps even shown there, that the realization is beneficial.

Cross-Cutting Requirements

| Usability | Performance/Reactivity | Good Fit to Context | Broad and Deep Support |

Core Features
- Determine the Changes that Need to be Reviewed
- Allow Viewing the Changes that Need to be Reviewed
- Collect, Store and Distribute Remarks
- Manage the Review Process

Cognitive-Support Code Review Tools

Advanced Reviewer Support
- Reduce Cognitive Load: Shrink the Task
- Reduce Cognitive Load: Help to Off-Load Items from Human Memory to the Computer
- Reduce Cognitive Load: Allow Efficient Chunking
- Take the Differing Working Styles and Backgrounds of Developers into Account
- Support Checking

Further Basic Features

| Allow Communication around Remarks | Allow Fixing on the Fly | Fixing Support for the Author |

Figure A.1: Overview of the essential requirements for cognitive-support code review tools

Weak deductive evidence. The realization is mentioned in the literature, or can be derived with additional assumptions.

A.1 Cross-Cutting Requirements

From the interviews and from the experiences gained when using CoRT in practice, I found a number of cross-cutting requirements that need to be taken into account throughout the whole tool. When these are not addressed adequately, developers resort to using general-purpose software development tools. This topic cropped up in the interviews (Part I), and is a possible reason for the high number of developers that do not use a specialized review tool (see Section 7.3).

A.1.1 Usability

As Myers et al. [278] state it, "Programmers are Users too", and good usability is a major concern to allow efficient use of software development tools. A code review tool should streamline the review process. Some examples where usability plays a role is the integration with the ticket system to avoid context switches (Section 9.1), adding of remarks directly from the source view (Section 9.2) and the presentation of diffs for efficient examination (Chapter 11).

A.1.2 Performance/Reactivity

The performance and reactivity of the review tool could be regarded as a sub-aspect of usability. Reviewers do not want to wait more than a few seconds when starting a review (see Section 15.3), and other interactions with the tools shall be swift, too. To reach this goal, it might be needed to cache data locally or to trade reviewer support for speed when the full calculation takes too long (see Section D.1).

A.1.3 Good Fit to Context

The development and review process and other contextual factors that influence the review tool differ a lot between teams and companies (see Chapter 6). A review tool has to integrate well into this context. Two examples: if there is an interface to an SCM, this needs to be the SCM used in the team; and if the team uses a post-commit review process this needs to be supported by the tool. For each contextual variant, a tool creator can decide whether to support it and make the tool configurable in this regard, or to not support it. In the latter case, the tool might not be usable for these teams, or the teams might decide to change the context (i.e., the tool shapes the process; see Section 6.2).

A.1.4 Broad and Deep Support

The variety in the tool's context also applies to the artifacts under review. There is not only source code in one programming language, but other languages, build scripts, test cases and much more that can be changed in a unit of work [120]. The advanced reviewer support that is proposed in this thesis is based on language-specific analyses, and such a deep support is only possible for a small part of the possible file types. But there needs to be a graceful fallback to keep up broad and robust support for all changes, for example resorting to simple textual diffs. A similar reasoning applies when there are very large changes that cannot be analyzed fully without compromising performance. Here, a graceful fallback is needed, too, to ensure basic functionality is still working.

A.2 Core Features

These are the core requirements that every tool for change-based review needs to satisfy. They are satisfied by most current tools (see Chapter 7).

A.2.1 Determine the Changes that Need to be Reviewed

A review tool must be able to determine the scope of the review for a unit of work, i.e., which parts of the code base need to be checked in the review.
Realization: Interface to SCM To determine the review scope, a review tool can gather the needed information automatically from the SCM. The tool determines the commits that belong to the unit of work and the changes that were performed in these commits. In this way, the tool is well integrated into the development context and provides a streamlined user experience. (⬚⬚ ⬚ ; Ch. 4, Ch. 7, Ch. 9, [241, 318])

A.2.2 Allow Viewing the Changes that Need to be Reviewed

The second core requirement is that the reviewer must be able to view and explore the changes that need to be reviewed with the review tool. There are several non-exclusive ways to allow this.
Realization: Show changes/diff To understand a change, it can be helpful to see the origin (old version) of the change. Checking that the new code works is easier when the reviewers know what changed and can assume that the old version did not contain unknown defects. To allow this type of analysis, a review tool can provide a two-pane diff view of the changes. Chapter 11 discusses a number of different ways to present these diffs.

(⬚⬚⬚ ; Ch. 7, Ch. 9, [296, 318])

Realization: Show in current code While working on a task, a portion of code may be changed several times. With post-commit reviews, changes might also happen from outside the unit of work under review. These intermediate steps do not matter much, the important question is whether there are issues in the final version of the code. To allow this type of analysis, a review tool can provide a view in which the review scope is highlighted in the most recent version of the code. Such a view also aligns well with other patterns such as 'Review in IDE editor' (Section A.4.2) or 'Allow for exploratory testing' (Section A.3.5).

(⬚⬚ ; Ch. 9)

Realization: Free Navigation The reviewer needs to explore the various parts of the change under review. If the reviewer has a good knowledge of the code base, a good approach can be to form hypotheses on expected changes and to check whether the changes were done as expected. A review tool can support this type of analysis by providing means for free navigation, with facilities such as hyperlinking, determining callers, determining subclasses, and so on.

(⬚⬚⬚ ; Ch. 9, Ch. 14, [140, 377, 387])

Realization: Guided Navigation The reviewer needs to explore the various parts of the change under review. With limited background information, it is hard to predict which parts of the code need to be checked and in which order. The review tool can help the reviewer by providing a means for guided navigation. In its simplest form, this can be a list of the change parts that need to be reviewed. Chapter 14 deals with the question how to determine a good order to guide the reviewer.

(⬚⬚⬚⬚ ; Ch. 9, Ch. 14, [157])

A.2.3 Collect, Store and Distribute Remarks

Another core requirement for a review tool is the handling of review remarks: Reviewers must be able to enter remarks, and these must be stored and distributed to the other stakeholders. Storing the remarks also allows for later analysis or auditing.

Realization: Adding of remarks in source view Reviewers usually notice an issue that leads to a remark while viewing the code. To streamline the process of adding a remark, the review tool can provide a feature to add a remark in such a view and to take the current context into account. For example, the remark could automatically be attached to the current line or current file.

(⬚⬚ ; Sect. 7.3, Ch. 9)

Realization: Remark classification Remarks may come in different types. For some, the reviewer thinks that they definitely need to be addressed, but others might be optional, or even just notes for the author. To allow the author to clearly distinguish these types of remarks, the review tool can force the classification of remarks. CoRT's remark types can be found in Section 9.4.

(⬚⬚⬚ ; Ch. 9, [140])

Realization: Remark storage in tickets Many ticket systems allow the addition of custom fields to tickets, and the review tool can use such a field to store the review remarks. In this way, they are automatically stored together with their ticket, and there is no need for a separate database. This also allows other developers to read the remarks without having to use the review tool, and therefore allows for gradual introduction and graceful fallback.

(⬚⬚ ; Ch. 9)

Realization: Remark storage in separate database When a review tool uses 'Remark storage in tickets', the review tool depends on the ticket system. This dependency might hamper its use in a broad range of teams with different ticket systems. Another possibility to store remarks is to use a separate database. This is available in CoRT as a second option, and is used in many other review tools.

(⚒️⚏🗒️ ; Sect. 7.3, Ch. 9, [141, 275, 338])

A.2.4 Manage the Review Process

Another core requirement is that the review tool must manage the review process, for example by allowing reviews to end with either acceptance or rejection, spawn the fixing of remarks, etc. The exact form of the support also depends on the chosen review process. The classification scheme in Chapter 6 presents many of the relevant process facets (e.g. Post-Commit vs Pre-Commit, Fixing on the same ticket or postponed to separate tickets, ...). Rule-based process integration is one of the factors that differentiate change-based review from other forms of review which rely on subjective review decisions, and Chapter 5 shows that such process integration is associated with a higher chance of keeping up the use of reviews.

Realization: Interface to Ticket System Often, the development process for tickets is managed in the ticket system. It can be adapted to also include the review sub-process. By interfacing with the ticket system, the review tool can find open reviews and change the state of tickets. This provides for a streamlined review experience while still leaving the possibility to gradually introduce the tool.

(✖🖼️ ; Sect. 7.3, Ch. 9)

Realization: Pull Requests Another option to embed reviewing in the development workflow are 'pull requests'. With pull requests, code changes are not directly applied to the main/master repository. Instead, submission of changes leads to a 'pull request' which is then reviewed (pre-commit review).

(🖐️⚏🗒️ ; Ch. 10, [147])

Realization: Team-wide configuration For regular reviews, the review process is defined for the whole team or company. Therefore, a review tool should allow the team-wide configuration of the respective settings. For teams that use a 'mono repository' strategy [184], this can be done by committing the configuration to version control.

(⚒️⚏ ; Sect. 4.1, Ch. 9)

A.3 Advanced Reviewer Support

The requirements stated in the previous section form the core of what today's review tools provide. Section 12.1 motivates that the main way to achieve better review results goes through the reviewer, either by finding (or educating) better or better-matching reviewers, or by supporting the existing reviewers to understand large changes better. The latter is more amenable for tool support. I propose that the review of large changes means high cognitive load, and that reducing this cognitive load can lead to better understanding and review results. The following requirements outline several ways to reduce the reviewer's cognitive load.

A.3.1   Reduce Cognitive Load: Shrink the Task

A cognitive-support review tool should reduce the size of the task, either by reducing the amount of code to review or by reducing the depth with which it needs to be reviewed. This not only reduces the cognitive load, but also directly reduces the needed effort.

Realization: Leave out unimportant change parts A cognitive-support review tool can reduce the review scope by identifying change parts that are not important for review and leaving them out from the list of items that need to be reviewed (see 'Guided Navigation' above). Chapter 15 performs a data mining study to classify change parts by importance. It concludes that simple syntactic checks are sufficient to reap most of the benefits here.

(👆 ✖ 🔢 📑 ; Ch. 15, [136, 373])

Realization: Help focusing on the most important parts A cognitive-support review tool can also help the reviewer to decide which parts of the change need to be checked in detail and for which parts skimming is sufficient. Section 15.2 touches upon this topic, but it is not studied deeply in the current thesis.

(📑 ; Sect. 15.2, [311])

A.3.2   Reduce Cognitive Load: Help to Off-Load Items from Human Memory to the Computer

The more items have to be stored in human working memory for a task, the higher its mental load. Therefore, a cognitive-support review tool should help the reviewer to off-load items from his or her memory to the computer.

Realization: Take Care of Tracking what was Checked Apart from the main task of checking the code, a reviewer has to perform book-keeping to assure that no important part of the change is missed. A cognitive-support review tool can provide means to move this book-keeping load to the computer. One possibility is to automatically track which parts of the change have been viewed, another is to allow the reviewer to explicitly mark change parts as checked.

(✖ 🔢 ; Sect. 9.2)

Realization: Allow Offloading of Temporary Markers to the Tool During review, a reviewer might suspect a potential problem, but needs to check other things first before confirming the suspicion. Keeping such a suspicion in memory adds to the mental load of the task. A cognitive-support review tool can provide temporary review markers that the reviewer can use to keep notes of such suspicions. In this way, the review tool can make sure that they are not lost.

(✖ 🔢 ; Sect. 9.2, Sect. 9.4)

Realization: Efficient guided exploration Frequent context-switches lead to unnecessary mental load. Reviewers often rely on the order given by the review tool to guide them through the change, especially when they are no experts in the respective part of the code base (Section 14.2). The theory developed in Chapter 14 of this thesis argues that an optimal order of reading source code keeps related change parts close together, which reduces the mental load. A cognitive-support review tool can automatically determine such an order and allow more efficient guided exploration that way.

(👆 ✖ 🔢 ; Ch. 14)

A.3.3 Reduce Cognitive Load: Allow Efficient Chunking

Another way to overcome the limits of human working memory is chunking. By chunking, a number of items are conceptually combined into a single item, and this combined item can be processed more effectively in working memory. An example is a developer that subsumes a large number of syntactic source code items as "a standard loop over the items in an array". A cognitive-support review tool should help the reviewer to perform mental chunking.

Realization: Provide background/context information Mental chunking depends on labels and links that connect the information fragments. A cognitive-support review tool can provide such background information, for example by showing the description of the task under review or the commit descriptions. The reviewer can then map the code changes to these descriptions more easily.
(⬡⬡ ; Ch. 14)

Realization: Provide views of the change with different granularity A cognitive-support review tool can also suggest groupings of the change. Having such high-level views of the change can help the reviewer to put the low-level changes in context. One such possibility is discussed in Section 14.3 with the hierarchical view of the code change. Another is Gripp's summary view of the change briefly discussed in Section 12.2.3.2.
(⬡⬡ ; Ch. 14, [150])

A.3.4 Take the Differing Working Styles and Backgrounds of Developers into Account

Cognitive load is based on the combination of the mental load of the task and the capabilities of the human. The human influence is not only a matter of raw cognitive abilities, but also depends on its background and experiences. For example, the needed support differs between a developer with a lot of experience in the changed portion of the code and another with little experience there. A cognitive-support review tool should allow different developers to benefit from its features.

Realization: Customizability In some cases, developers know what way of presenting information is best for them. And if they don't, providing ways to customize the tool to their wishes can still increase the tool's acceptance. A cognitive-support review tool can provide customizability in various regards, for example for the creation of the hierarchical tours (Chapter 14) or for the presentation of the source code diffs (Chapter 11).
(⬡⬡⬡ ; Ch. 11, Ch. 14, [278])

Realization: Efficient free exploration Developers with experience in a code base might readily know how a certain method is usually used, whereas others need to collect this knowledge from the code on-demand. A cognitive-support review tool can provide various means for free exploration of the change and the code base: Hyper-linking and back-referencing between callers and callees, analysis of inheritance hierarchies, tooltips with Javadocs, and so on. Many of these features can be subsumed as IDE-like navigation features, and consequently one possibility to provide these features is by integrating the review tool with an IDE.
(⬡⬡⬡ ; Ch. 9, Sect. 12.2.3.2, [323, 361, 362])

A.3.5 Support Checking

This thesis focused on more efficient understanding of large changes, mainly by reducing the cognitive load of the reviewers. But this is not the only way to aid the reviewer, support can

also be provided for the checking itself. The thesis briefly touched upon two ways to do so.

Realization: Incorporate review agents / context-sensitive checklists Context-sensitive checklists, i.e., checklists whose content depends on properties of the change, can help to find problems during check-in [28]. They are similar to the review agents that were touched upon in Section 15.2. Both analyze the change and create items of the form "There could be this kind of problem here. Please check it." Similar checklists can also be used during review, to provide hints especially for inexperienced reviewers. It remains to show in future work whether a framing as checklist targeted at the reviewer or as agent-created remarks targeted at the author is better.

(⛏📑 ; Sect. 6.1, Sect. 15.2, [28, 69, 121, 280])

Realization: Allow for exploratory testing The interviews and the survey in Part I show that developers sometimes execute the code during review, for example to gain a better understanding or to test a hypothesis about a potential problem. A review tool can allow such execution, for example by integrating into the IDE.

(⚒⛏ ; Sect. 6.1)

A.4 Further Basic Features

So far, this section discussed the core features that every change-based review tool must possess and the cognitive support features that are one of the main contributions of this thesis. But apart from these, there are more features that a creator of a review tool should consider.

A.4.1 Allow Communication around Remarks

Reviews are not only about finding defects, but also about communication and discussion of improvement possibilities (see Section 4.3). Like other remarks, this discussion usually spawns at a certain position in the code, or as a reaction to a remark.

Realization: Discussion Threads for Remarks A review tool should allow discussion threads for remarks, i.e., other developers can respond to remarks and this responses form a linear discussion thread. By starting at a remark, the discussion has a focus and is bound to a certain position in the code, and the linear form is easy to grasp.

(⚒⛏📑 ; Sect. 6.1, [318])

A.4.2 Allow Fixing On-The-Fly

Chapter 6 mentions that many developers commonly fix small issues on-the-fly during review. A review tool should support such on-the-fly edits. The reviewer should still be able to give a rationale for the change, and other reviewers or the author should be able to see and check them.

Realization: Review in IDE editor The review tool can provide the possibility to fix remarks on-the-fly by integrating into the IDE. In this way, continuous compilation and many of the other checks in the IDE prevent the introduction of simple errors, and the changes need to be committed and are, therefore, automatically traceable to the reviewer. Chapter 9 discusses the pros and cons of integrating a review tool with the IDE in more depth.

(⚒⛏ ; Sect. 6.1)

A.4.3 Fixing Support for the Author

Without fixing the found issues, many of the benefits of reviews cannot be reaped. Therefore, a review tool should also provide support for the author during fixing.

Realization: List of Remarks By providing a list of the found remarks, the review tool can help the author to address them in a systematic way.
(⬛🗄 ; Ch. 9)

Realization: Navigation from Remarks to Code For remarks that are bound to a certain position in the code, the review tool should allow easy navigation from the list of remarks to that part of the code.
(⬛🗄 ; Ch. 9)

Realization: Check-off for Remarks The author needs to keep track of which items still need to be addressed. The review tool can provide a feature to check-off remarks as addressed or to reject their fixing.
(⚒🗄 ; Sect. 6.1, Ch. 9)

$$\diamondsuit$$

Summing up, this chapter presented essential requirements for cognitive-support code review tools, and possible realizations to satisfy these requirements. The requirements are separated into four groups, of which three apply to all code review tools and one contains the advanced reviewer support features that are specific to cognitive-support code review tools.

B

The Faceted Classification Scheme in Detail

This chapter presents the details on the facets of the classification scheme that was introduced in Chapter 6.1. When there was a corresponding question in the online survey from Part I, the percentages of the teams that use the respective variant are also given.

A variation point is included as a facet of the classification scheme when it is an identifiable, fixed part of the process for at least one case and at least two different variations were observed. Some of the interviewees reported several distinct cases of review use for a single company. For the scheme, I consider a review variant a distinct case if it differs from other review process variants in the same company and if this choice only depends on external factors (team, product, ...). The IDs from Table 3.1 (in Section 3.1) are used as subscripts to refer to the companies. When the ID is followed by a number, this refers to a specific case for that company.

B.1 Process Embedding

The first group of facets contains aspects that varied with regard to how the code review process is embedded into the rest of the development process, i.e., when and in which way a review is triggered and which influences it has on other process activities.

Unit of work The definition of regular change-based code review (Definition 2) states that the changes performed in a "unit of work" define the scope of the review. The smaller the unit of work, the smaller the effort spent on every single review, but the higher the number of reviews. In the studied cases, one of these types of unit of work was chosen as a trigger for reviews:

RELEASE Review is triggered for changes that are put into production or are ready for "production approval".$_{I.IQ}$ Survey: 1% (1/136)

STORY/REQUIREMENT Review is triggered for a user story/requirement that is considered done. $_{I.IF1,IH,IL}$ Survey: 11% (15/136)

TASK Many teams divide user stories into separate implementation tasks. If the chosen value for unit of work is "task", a review spans the changes done in such an implementation task.$_{I.IE,IF2,IG,LX,LF}$ Survey: 53% (72/136)

PUSH/PULL/COMBINED COMMIT Review is done for each source code management (SCM) 'pull request' or some other type of combined commit. A pull request often corresponds to a task when a team uses both. $_{I.IB1,IO,IS,LA,LS,LQ}$ Survey: 27% (37/136)

SINGULAR COMMIT Review is triggered for every small-grained SCM commit. Mainly, this variant is used when changes are rare and strictly controlled, such as in release branches. $_{I.ID,IB2,}$ $_{IP1,IP2}$ Survey: 8% (11/136)

Tool support/enforcement for triggering The triggering of reviews can either be supported by tools, or it is done completely manually:

TOOL A tool ensures that a review candidate is created for each unit of work. This can be accomplished, for example, with a separate state in a bug tracker's ticket workflow or with specialized tools that enforce this process, like pull requests on GitHub[1] or Gerrit[2]. $_{I.ID,IE,IF1,IF2,IG,IO,IS,LG,LS,LQ}$

CONVENTIONS If no tools are employed, conventions and group pressure are used to reach process compliance: "...and this is more like peer pressure. If something goes live and there was no review, the other developers will ask 'Why not? What's up?'"$_{I.IQ}$ $_{I.IB1,IB1,IH,IL,IQ,LM}$

Publicness of the reviewed code The changes under review can be made public to other developers before the review, or they are still private during the review. This is studied in detail in Chapter 10. The two corresponding values for this facet are:

POST-COMMIT REVIEW A review is performed after the changes are visible for other developers[3]. $_{I.IB1,IE,IF1,IF2,IG,IH,IL,IP2,IQ}$ Survey: 54% (76/140)

PRE-COMMIT REVIEW A review is performed before the changes reach the main development trunk[4]. Review using pull requests is a special case of pre-commit review. $_{I.IB2,ID,IO,IP1,IS,}$ $_{LA,LV,LM,LG,LQ,LN,LY,LH,LD}$ Survey: 46% (64/140)

Means to keep unreviewed changes from customer releases All studied teams that perform regular reviews to detect defects try to avoid performing reviews after the changes have been released to the customer(s). They choose different means to reach this goal:

ORGANIZATIONAL They manually check for open reviews when a release approaches, in combination with organizational means to ensure swift completion of reviews (see "Means to ensure swift review completion"). $_{I.IE,IF1,IF2,IH,IL,IQ}$ Survey: 26% (26/99)

PRE-COMMIT REVIEW They use pre-commit reviews (see "Publicness of the reviewed code"), either generally or for a certain time before releases. $_{I.IB2,ID,IO,IP1,IS,LA,LV,LS}$ Survey: 40% (40/99)

RELEASE BRANCH There is a permanent technical separation between development branch/stream and release branch/stream. $_{I.IG,LS}$ Survey: 33% (33/99)

---

[1] https://github.com
[2] https://www.gerritcodereview.com
[3] also called "Commit Then Review" in other publications
[4] also called "Review Then Commit" in other publications

Means to ensure swift review completion To ensure that a review does not stay open for too long and open reviews don't accumulate too much, many teams employ at least one of the following organizational means:

PRIORITY Reviews have higher priority than other tasks ("*In one team we defined a rule in which order tasks have to be worked on. That production problems are the first priority, but that when the implementation of a user story is done and it is ready for review on the taskboard, that this has a higher priority then starting a new story. To keep the cycle times in a sprint short.*"$_{I.IF}$). $_{I.IE,IF1,IF2}$ Survey: 24% (24/99)

WIP LIMIT The team has a "work in progress (WIP) limit"[5] that restricts the number of tasks that can be "ready for review". $_{I.IH}$ Survey: 13% (13/99)

TIME SLOT The reviewers reserve specific times of the day or week for code review. $_{I.IG}$ Survey: 19% (19/99)

AUTHOR'S RESPONSIBILITY The author actively seeks out a reviewer, perhaps they even review together. When pre-commit reviews are used, the author has an incentive to get the review done, in order to get his changes included into the common code base. $_{I.IB1,ID,IL,IO,IP1,IQ,IS}$ Survey: 35% (35/99)

Blocking of process There are steps following code review in a unit of work's life cycle. Commonly this is declaring the feature ready for use and delivering it to the customer. The observed cases differ in whether these following steps are blocked while the review is underway:

FULL FOLLOW-UP The steps following code review in a unit of work's life cycle, e.g., declaring it as 'ready for delivery', are not begun until all issues found in the review have been resolved. $_{I.IB2,ID,IE,IF1,IF2,IG,IL,IO,IP1,IP2,IS,LA,LF,LM,LG,LS,LQ}$

WAIT FOR REVIEW The unit of work is blocked until the reviewers finished checking. Fixing is done based on trust only and not explicitly waited for. $_{I.IH,IQ}$

NO BLOCKING The unit of work can be further processed, e.g., delivered to the customer, without checking for missing reviews. $_{I.IB1}$

B.2 Reviewers

The facets belonging to this group describe differences we found regarding the selection of reviewers.

Usual number of reviewers It was noted as a commonality of the observed processes that there are usually at most two reviewers (excluding the author). Some teams report rare cases with more reviewers: "*When it's database migration code [...], five to six selected reviewers have to give their OK to it*"$_{I.IH}$. In cases *ID* and *IL*, the author usually takes part in reviewing (see "Interaction while checking").

Rules for reviewer count / review skipping The teams have different rules regarding the minimal number of reviewers, and whether this number can be zero, i.e. review can be skipped. In most cases, at least one of the following factors can influence the minimal number of reviewers:

---

[5]a term from "Kanban" and similar methodologies [212]

COMPONENT Review is only obligatory for certain components, or certain components have to be checked by more reviewers. These components are commonly more complex or the estimated consequences of defects are more severe. *I.IB1,IE,IO,LS* Survey: 27% (25/93)

AUTHOR'S EXPERIENCE Only changes by inexperienced developers have to be reviewed. *I.IP2* Survey: 23% (21/93)

LIFE CYCLE PHASE In some teams, review is only obligatory near releases. *I.IB2,ID,IP1,LS* Survey: 12% (11/93)

CHANGE SIZE Changes smaller than a certain threshold do not have to be reviewed. *I.IL* Survey: 13% (12/93)

PAIR PROGRAMMING Changes done using pair programming do not have to be reviewed or have to be reviewed by one reviewer less. *I.IE,IF2,IS* Survey: 35% (33/93)

REVIEWER'S CHOICE The reviewer can decide that a review is not needed or that a second reviewer would be advisable, for example, based on an assessment of the change's complexity. *I.IF1,IF2,IG,IH* [6]

AUTHOR'S CHOICE The author can choose to have additional reviewers, e.g., because she considers the change risky. *I.IB1,IB2*

Reviewer population There are large differences in the studied cases regarding the set of potential reviewers. The significant factor is how, and to what extent, the experience of the potential reviewer is taken into account. The authors themselves are not employed as the only reviewer in any of the cases, but they are sometimes asked to review jointly with another developer. Among the studied teams, the variants were:

EVERYBODY Every team member shall be available as a reviewer for every change, mostly with some exceptions allowed. *I.IB1,IB2,IE,IF1,IF2,IH,IL,LF* Survey: 87% (80/92)

RESTRICTED Only a certain subset of experienced developers, core team members or specialists shall do reviews. *I.IG,IS,LF,LG,LL* Survey: 2% (2/92)

FIXED All reviews are done by the same reviewer(s). In one of the observed cases, the team elects two experienced reviewers to perform every review. Another variant is the team leader reviewing everything. *I.ID,IP2,IQ,LX* Survey: 11% (10/92)

Assignment of reviewers to reviews The interviewees described several possibilities how the connection between reviewers and reviews can be determined:

PULL Using this style of reviewer assignment, it is the reviewer who chooses among the outstanding reviews. *I.IE,IF1,IF2,LF* Survey: 30% (47/155)

PUSH This is the opposite of the "pull" style: The author chooses who should perform the review. *I.IB1,IB2,IL,LA,LV,LM,LQ,LL* Survey: 35% (54/155)

MIX In a mix of the "push" and "pull" style, the author invites a preferred subset of the reviewer population and the potential reviewers then decide if they want to participate. *I.IG,IH,IS,LA, LM,LQ* Survey: 21% (32/155)

FIXED The same reviewers perform all reviews for a certain team or module, so that there is no choice left. *I.ID,IP2,IQ,LA,LV,LQ* Survey: 12% (18/155)

DISCUSSION The assignment of reviewers is discussed in the team. Survey: 1% (2/155)

RANDOM Reviews are assigned randomly to reviewers. Survey: 1% (1/155)

---

[6]The options "reviewer's choice" and "author's choice" were not considered in the survey, because the survey focused on situations where a specific number of usual reviewers can be given.

Tool support for reviewer assignment When the number of potential reviewers is large and reviewer assignment is performed in the 'push' style, tool support can help the authors find suitable reviewers.

NO SUPPORT There is no support for finding a suitable assignment of reviewers to reviews. *I.ID,IE, IF1,IF2,IG,IH,IL,IP2,IG,IS*

REVIEWER RECOMMENDATIONS There is computer support for finding suitable reviewers for a given change. *I.IB1,IB2,LV,LF,LM,LN,LD*

Tool support for 'pull' assignment, i.e., help for the reviewers to find open reviews that are suitable for them, comes to mind as an obvious additional possibility. We could not observe this kind of support in the studied cases.

B.3 Checking

The facets in this group describe variations that were observed regarding the central activity of a code review: Checking the code. Like in the rest of this chapter, I only describe variations in the codified processes of the teams. Personal differences in the checking habits of the individual reviewers are out of the scope of the classification scheme.

Interaction while checking Differences in reviewer interaction while checking have been described in some of the earliest publications on code reviews. We could also observe similar differences in the studied cases:

ON-DEMAND The review participants only interact on-demand, e.g., when there are questions regarding the code. *I.IE,IG,IP2,LV,LF* Survey: 16% (26/158)

ASYNCHRONOUS DISCUSSION In addition to on-demand interaction, review remarks are instantly communicated and are discussed asynchronously by the review participants. *I.IB1,IB2,IH,IO, IS,LA,LM* Survey: 56% (89/158)

MEETING WITH AUTHOR All review participants, potentially including the author, meet for checking. Often, the author actively guides the reviewer and explains the code. This has been called "pair review" by some of the interviewees. *I.ID,IF1,IF2,IL,LS* Survey: 26% (41/158)

MEETING WITHOUT AUTHOR The reviewers meet to discuss the code. This variant has been called "pair review", too. *I.IQ* Survey: 1% (2/158)

The main use of direct interaction, according to the interviewees, is to help to understand the code and to gain background information on implementation decisions (e.g., to decide if something is there for a legitimate reason or if it is an issue). It is also used to foster discussion on better solutions.

Temporal arrangement of reviewers When multiple reviewers perform a review, we observed two variants of temporal arrangement:

PARALLEL All reviewers work in parallel and rework starts when all are finished. Parallel review is a prerequisite for direct inter-reviewer interaction. *I.IB1,IB2,ID,IF1,IF2,IH,IO,IQ,LF,LM* Survey: 73% (73/99)

SEQUENTIAL Only one reviewer reviews at a time and rework is done after each reviewer. *I.IE,IG,LV* Survey: 26% (26/99)

Specialized roles   When there are multiple reviewers, they can take on different roles, specializing on certain quality aspects. We only observed a limited number of cases using different roles, so that we divided this facet's possibilities coarsely:

ROLES The different reviewers take on different roles, e.g., with one reviewer specializing in code quality and another looking at GUI aspects. $_{I.IG,LF}$ Survey: 7% (6/90)

NO ROLES Reviewers do not take on different roles. $_{I.IB1,IB2,ID,IE,IF1,IF2,IH,IL,IP2,IQ,IS,LF,LQ}$ Survey: 93% (84/90)

Detection aids   The checking in a code review is performed mainly by viewing and reading source code (Definition 1). To increase the chance of finding defects, some teams use one or several auxiliary techniques:

CHECKLISTS A checklist, i.e., a list of questions or topics that have to be checked in a review, is used by a subset of the observed cases. $_{I.IF1,IF2,IL,IQ,IS,LA}$ Survey: 23% (22/94)

STATIC CODE ANALYSIS Static code analysis *during* checking can be used to guide and focus the reviewer on portions of the code that might need further attention. This differs from the use of static analysis before reviewing that is commonly done to find simple issues early (see Section 4.1 and Chapter 15). $_{I.IE}$ Survey: 79% (72/91)

TESTING In several of the cases, a limited amount of manual exploratory testing is common when performing a review (sometimes even adding additional unit tests during reviewing). $_{I.IB1,}$ $_{IB2,IE,IF1,IF2,IG,IL,LS}$ Survey: 76% (69/91)

Reviewer changes code   When performing a code review, it is often tempting for the reviewer to directly change some code. The observed teams use one of two options with regard to code changes by the reviewer:

NEVER The reviewer is not allowed or technically not able to change code during checking. $_{I.IB1,}$ $_{IB2,ID,IH,IQ}$ Survey: 46% (73/157)

SOMETIMES The reviewers may change code during checking. This option is most often used for changes that are small and which are regarded as risk-free and not worth the effort to write a remark. $_{I.IE,IF1,IF2,IG,IL,IS}$ Survey: 54% (84/157)

B.4   Feedback

Feedback in the form of review remarks is the main result of the checking. The facets in this group summarize differences in the handling of feedback between the teams' review processes.

Communication of issues   The bulk of issues that is not fixed by the reviewer (and sometimes also issues that have been fixed) has to be communicated to the author. This communication can generally be divided into oral and written communication.

WRITTEN The found issues are mainly communicated in written form. Depending on the used tools, this can happen for example in the form of email, as ticket comments, using comments in the source code or using a specialized review tool. $_{I.IB1,IB2,IE,IF1,IF2,IG,}$ $_{IH,IO,IQ,IS,LA,LV,LF,LM,LG,LS,LQ}$ Survey: 53% (84/158)

ORAL ONLY The issues are discussed orally with the author, mostly face-to-face. They are not stored except for short-term note taking. $_{I.ID,LX}$ Survey: 23% (36/158)

ORAL STORED The issues are mainly communicated orally, but also stored in written form, e.g., for traceability purposes. $_{I.IL}$ Survey: 24% (38/158)

Options to handle issues When the review remarks finally reach the author, he or she decides how to cope with them. The following possibilities were found:

RESOLVE Change the code according to the remark. Survey: 100% (91/91)

REJECT Ask for clarification or justification of the remark (when it is seen as some sort of 'false positive' by the author). Survey: 99% (90/91)

POSTPONE Create a task to perform the needed changes. This task is then prioritized according to the team's development process. $_{I.IB1,IH,IL,IQ}$ Survey: 65% (59/91)

IGNORE Do nothing. $_{I.IB1,IH,IQ}$ Survey: 42% (38/91)

The actually found differences apply to the options POSTPONE and IGNORE. Some teams dismiss postponing with an attitude of "either fix it now or forget it". And some teams demand a reaction on every raised issue ("*...when the reviewer says something, that either has to be done this way or it has to be discussed together and a consensus reached.*" $_{I.IS}$) while for some it is an accepted practice to ignore remarks. The survey also asked about the frequency of the respective behaviors. When only counting teams in which the respective behavior happens occasionally or more often, the counts go down to 31% (28/91) for "postpone" and 4% (4/91) for "ignore".

B.5 Overarching Facets

The facets in this group pertain to aspects that span the whole code review process.

Use of metrics The collection of review metrics and the systematic use of review results to improve the process is mentioned in the literature as a key feature that separates "inspection" from "Inspection" [140]:

METRICS IN USE Metrics on the code reviews are gathered and used systematically, for example for report generation and process improvement. $_{I.LA,LF,LM}$ Survey: 5% (4/88)

NO METRICS USE Metrics on the code reviews are either not available or not used systematically. $_{I.IB1,IB2,ID,IE,IF1,IF2,IG,IH,IL,IO,IP1,IP2,IQ,IS}$ Survey: 95% (84/88)

We could not observe the value METRICS IN USE in the empirical data from the interviews, but only in the cases described in the literature and in the survey.

Tool specialization In the preceding sections, several facets referred to tool support for specific tasks. More generally, we observed a difference between the use of a specialized tool and the combination of general-purpose tools:

GENERAL-PURPOSE No specialized review software is in use. Instead, the teams use a combination of IDE, source code management system (SCM) and ticket system/bug tracker. Combined, these systems support all features seen as the core of a review tool: Viewing, examining and possibly editing the code, determining which parts of the code belong to a unit of work, documenting issues and integration of reviews into the development process. $_{I.ID,IE,IF1,IF2,IG,IL,IQ}$

SPECIALIZED The teams use a specialized code review tool that combines all relevant features. $_{I.IB1,IB2,IH,IO,IP2,IS,LA,LV,LF,LM,LG,LS,LL}$

Section 7.3 goes into more detail on the specific tools that are used in the sample.

$$\diamondsuit$$

By choosing a faceted description, it is possible to cleanly describe the variations found in the study. A weakness of this approach is that it is more complicated to use, compared to a small number of placative process names. In addition, interdependencies between certain values of the facets are not immediately obvious. Another drawback of the proposed model is the large number of facets. There intentionally was no exclusion of facets based on some measure of relevance, because relevance depends on the context in which the model is used. Researchers using the model should consider excluding facets that they know to be irrelevant for their study.

C

Details on the Simulation Model for the Comparison of Pre- and Post-Commit Reviews

This chapter provides implementation details on the discrete-event simulation model that is used in the simulation study in Chapter 10. The description is based on [35].

The model's basic structure follows an agile Kanban-style [9] development process, incorporating agile best practices and common industrial practices: A development team continuously pulls user stories from a pool of requirements, splits them into tasks, works on the tasks until all of them are finished and then delivers the results to the customers. After the implementation of a task is finished, the corresponding changes are reviewed. If issues are found during the review, they have to be fixed and another round of review follows, otherwise, the task is considered done. This basic life-cycle of a task is depicted in Figure C.1[1]. Further quality assurance measures, like unit testing and static code analysis, are subsumed under "implementation". The model assumes that some kind of 'continuous delivery' process ensures that finished stories will be available to the customer shortly after they are finished, but does not model it in detail. Also out of scope is the prioritization process: Stories are simply created with random attributes when needed.

At the start of Chapter 10, the effects that the interviewees believed to influence whether

---

[1]As advised in [291], the UML is used to describe the model. The full diagrams are available online [33].
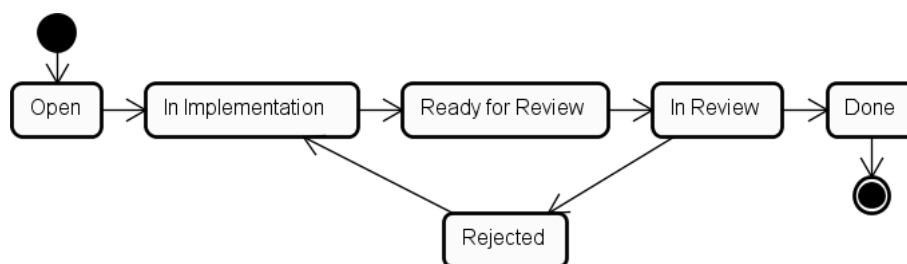
Figure C.1: State diagram for *Tasks* (see also Figure C.2) (Source: [35])
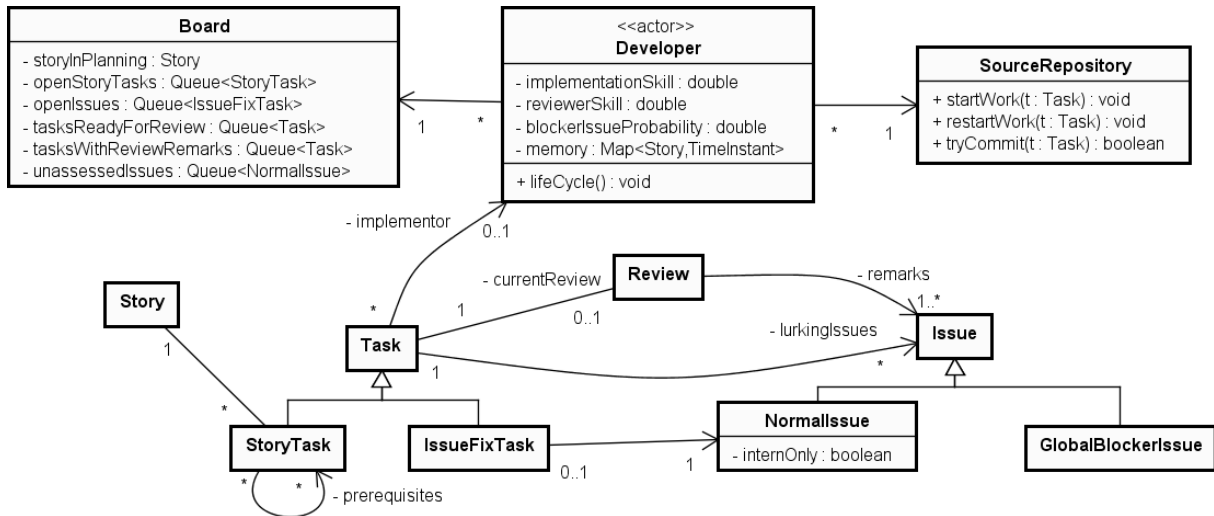
Figure C.2: Overview of important classes from the model (Source: [35])

pre-commit or post-commit reviews are a better fit are shown. All of these effects are considered in the model: Injection of 'normal' and 'blocker' issues, task switch overhead, and conflicts on commit. The review variant is a special model parameter that is used in some of the sub-processes described below.

Figure C.2 shows the classes constituting the model. It contains two 'active' parts: *Developers*[2] and *Issues*. The next sections first describe a *Developer's* life-cycle in more detail and elaborate on *Issues* afterward. Not every detail of the model is described as thoroughly as needed for detailed replication and scrutiny of the results. Therefore, the full model has been made available online [33].

C.1 Details on the Modeling of Developers' Work

A *Developer* is an active process and follows the process model depicted in Figure C.3. When looking for something to do, the *Developer* follows a strict priority order: When there is an *Issue* that just newly occurred, she has a look at it and decides what to do with it ("issue assessment"). When no such *Issues* exist, she checks whether a *Task* she implemented was rejected with review remarks and fixes these remarks. Next in priority order comes the reviewing of *Tasks* implemented by others, then fixing of *IssueFixTasks*, then implementation of new *StoryTasks* and finally, when there is no other work to do, planning a new *Story* or joining someone else who is already planning.

The sub-process of implementing a task belonging to a story is shown in Figure C.4. It starts with the *Developer* changing the *Task's* state to "in implementation" and moving it to the corresponding column of the *Board*. She updates her local working copy from the source code repository, which registers the current simulation time for the later calculation of potential conflicts. The following two activities are shown separated, but could also be seen as interleaved: Depending on the time she last worked on the *Story*, she has to spend some "task switch overhead" to (re-)familiarize herself with it. Additionally, she performs the implementation work (including testing), which for the sake of this model mainly boils down to the creation of

---

[2]In the rest of this section, class names are written in *italic*.

194
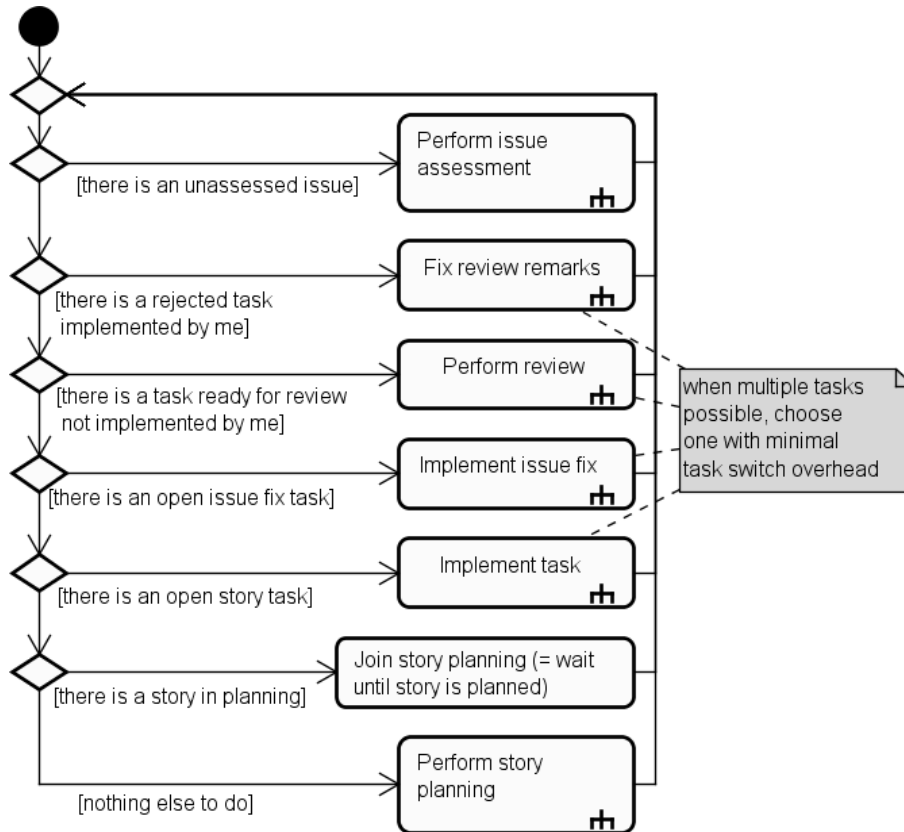
Figure C.3: Activity diagram: Main process followed by *Developer* (see also Section C.1) (Source: [35])

new *Issues*. The step after that depends on the review mode: In pre-commit review mode, the *Task's* state is directly changed to "ready for review". In post-commit mode, she first has to commit her changes, during which there can be conflicts that have to be resolved.

When performing a review, the *Developer* first changes the *Task's* state and has to cope with task switch overhead, too (see Figure C.5). Then, the code is checked, which means that each *Issue* that is currently 'lurking' in the *Task* is found with a certain developer-specific probability. If the developer found at least one *Issue*, she rejects the *Task* and notes the found *Issues* as review remarks. Otherwise, if in pre-commit mode, the *Task's* accumulated changes will be committed to the main development source tree, like described above. The *Task* is then considered "done", which could mean that the corresponding *Story* can be finished.

The process for fixing review remarks is similar to the process for the implementation of tasks. One of the main differences is that when fixing review remarks, the corresponding *Issue* will be marked as "fixed" on commit so that it cannot occur anymore (see Figure C.6 for the issues' states). Furthermore, the time needed for fixing depends on the sum of the single remarks' sizes and not the task's size.

C.2   Details on the Modeling of Issues

The preceding section already alluded to the life-cycle of an *Issue*, which is now described in more detail: In the model, an *Issue* is very broadly defined as every true positive that can
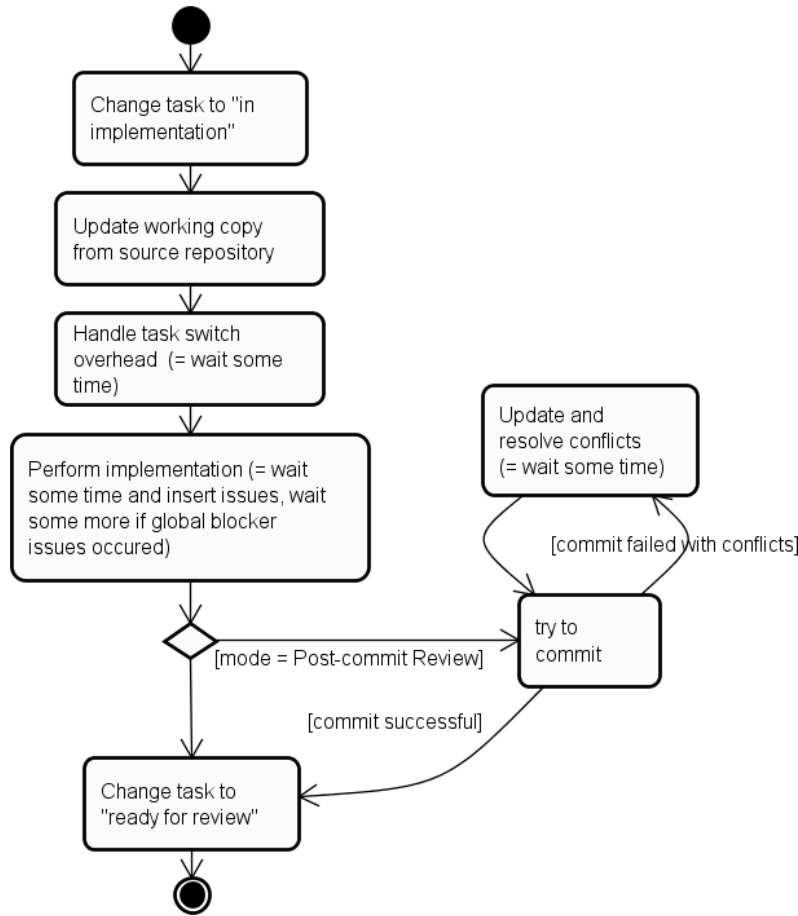
Figure C.4: Activity diagram: Details for the sub process "Implement task" (from Figure C.3) (Source: [35])

be remarked in a review [248]. The model distinguishes two kinds of issues: *NormalIssues* stand for problems like an incorrectly implemented algorithm (corresponding to the definition of fault in [178]), as well as maintainability or performance problems. *GlobalBlockerIssues* on the other hand handle the special case when a *Developer* injects a problem that will block all other currently implementing *Developers* soon after it is committed. In contrast, when a *NormalIssue* is observed by a developer or customer, it is put in a queue with *Issues* that have to be assessed. During issue assessment, a *Developer* determines which *Task* the *Issue* belongs to. Depending on the *Task's* current state, the *Issue* can be fixed by some *Developer* who is currently at it anyway, or an *IssueFixTask* has to be created. Conceptually, time lost due to distraction by an issue while working on another task is also modeled as part of the issue assessment time. Further details on issue assessment can be seen in Figure C.7.

There are two ways a *NormalIssue* can become visible. It can be observed by a *Developer* after it has been committed to the main development source tree, and it can be observed by a customer after the story has been finished and delivered (see Figure C.6). As soon as a fix for the *Issue* is committed, it cannot be observed anymore. Not all *NormalIssues* can be found by customers: A significant share of issues found in code reviews are maintainability issues [248], and as such can only be observed by *Developers* ("internOnly").

196
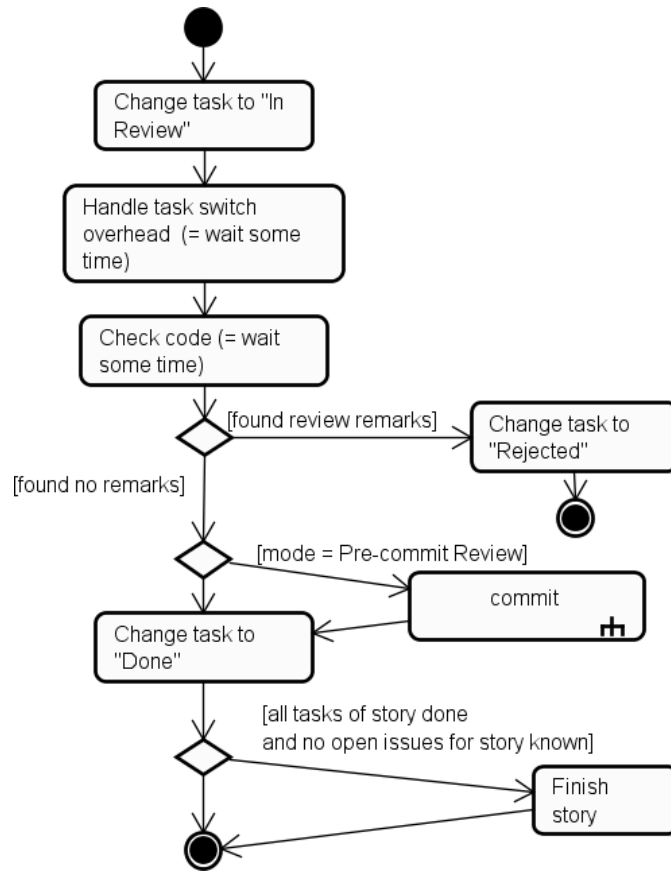
Figure C.5: Activity diagram: Details for the sub process "Perform review" (from Figure C.3) (Source: [35])
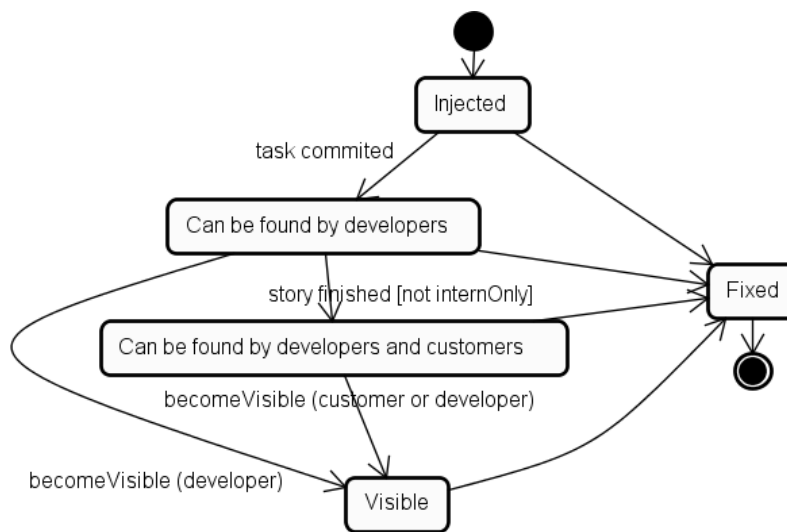
Figure C.6: State diagram for *NormalIssues* (see also Figure C.2) (Source: [35])
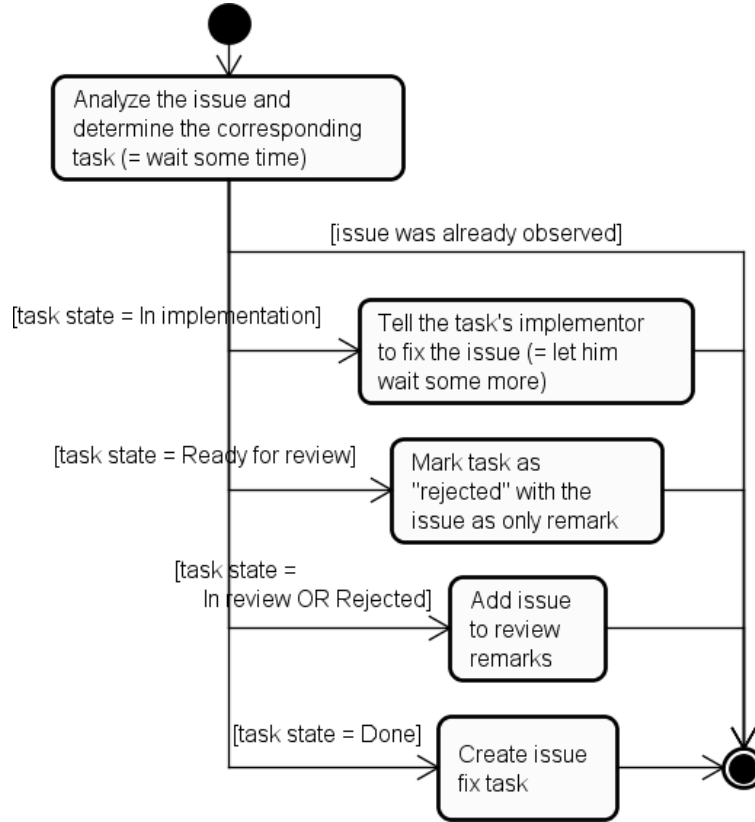
Figure C.7: Activity diagram: Details for the sub process "Perform issue assessment" (from Figure C.3) (Source: [35])

Issues can be injected every time a *Developer* implements a *StoryTask*, or fixes a review remark or an *IssueFixTask*. The number of *Issues* injected is calculated based on the *Developer's* skill (measured in issues/hour for the sake of simplicity), and the time spent implementing (taken as a surrogate for the task's complexity). An additional increase is due to follow-up issues[3]:

$$\text{issue count} \approx \text{issue injection rate}_{\text{developer}} \cdot \text{relevant time}$$
$$+ \text{follow up issue spawn probability} \cdot \sum_{p \in \text{prerequisites}} \left| \text{lurking issues}_p \right| \quad \text{(C.1)}$$

When fixing issues, there is an additional 'fixing issue rate factor' in the first part of the sum. It accounts for the possibly lower chance of introducing new issues when fixing old ones (due to fewer degrees of freedom). The relevant time depends on several parameters and on the type of the task. As an example, it is calculated for issue fix tasks as:

$$\text{relevant time} := \text{review remark fix duration} \cdot \text{review fix to task factor}$$
$$+ \text{task switch overhead} \cdot \text{task switch issue factor} \quad \text{(C.2)}$$

The calculation of the task switch time is modeled with an exponential decay, inspired by the forgetting curve of Ebbinghaus [243]. Full details on its calculation are available in the model [33].

---

[3]Please refer to Tables C.1 and C.2 and Figure C.2 for more details on the parameters and terms from the model used in the formulas. Like the rest of the model, the formulas were generated and validated with the iterative process described in Section 10.1.2

C.3 Empirical Triangulation of Model Parameters

As described in Section 10.1, the main empirical foundation of the simulation study is qualitative. In addition, quantitative empirical data was used in three ways: (1) To validate the shape of probability distributions for input parameters, (2) to determine realistic dependency structures, and (3) to cross-check the choice of input parameter ranges. The quantitative data was collected in retrospect from the partner company's ticket system. It contains information on all software development tasks for the company's product for several years.

The distribution for the time between when an *Issue* could be observed and when it will be observed belongs to the category (1) ("determine the shape of probability distributions"). I sampled 15 user-found issues and manually determined the date they were injected and the date they were observed (i.e., the bug ticket was created). The median difference was 207 days. The obtained distribution matched an exponential distribution fairly well. As it can also be argued on a theoretical basis that the time until an issue is found by a customer follows a (shifted) exponential distribution, this distribution was chosen for the simulation.

To cross-check the choice of parameter ranges for the durations of task implementation and review, I also sampled data from the same ticket system. As this analysis was fully automated, I was able to take a larger sample. I excluded tasks with obviously erroneous durations, leaving 1896 tasks. The median time for implementation was about 4 working hours, the arithmetic mean about 9 working hours. For the review time, the median was 0.5 working hours and the arithmetic mean was 1 working hour. A day was counted as 8 working hours for the calculation. Both distributions are skewed to the right. The Pearson correlation coefficient of implementation and review time is (only) 0.18, so that I chose independent distributions for the implementation and review time. For distribution fitting, I compared log-normal, gamma, normal and exponential distributions using R [307]. A log-normal distribution is the best fit for all task durations.

As described in the previous section, new stories are created when needed in the simulation. During creation, the number of tasks and their dependency structure has to be determined. To gain realistic structures, I sampled 49 stories from the industrial ticket system and determined the dependency structure of the subtasks. The minimal number of tasks per story was one, the maximal number in the sample was 16. Small numbers of tasks were much more frequent than larger numbers. The obtained empirical distribution of task counts can be seen in Figure C.8. I encountered 25 different dependency graphs. To provide a little more insight into their structure, I determined the number of start tasks (tasks not dependent on anything) and end tasks (task on which no other tasks depends). In 14 of the graphs (15 of the stories), there are more end tasks than start tasks. In 4 cases it is the other way around so that there is a tendency towards common prerequisite tasks followed by parallel work. In addition to this "REALISTIC" dependency structure, the simulation study uses four artificial structures: In "NO_SUBDIVISION" every story consists of only one task. In "NO_DEPENDENCIES" the number of tasks per story is distributed like in "REALISTIC", but without any dependencies between the tasks. In "CHAINS" the tasks are totally ordered so that their dependencies form a sequential structure. Every story contains at least two tasks, otherwise the distribution of small to large tasks is similar to "REALISTIC". In the last studied structure, "DIAMONDS", there is a single start task and a single end task. All tasks in between can be done in parallel. Consequently, every story consists of at least three tasks. During sensitivity analysis, it turned out that the exact dependency structure is not that relevant so that only "REALISTIC" and "NO_DEPENDENCIES" were studied in full detail. More information on the graph structure

data can be found in the class *DependencyGraphConstellation* of [33].

Tables C.1 and C.2 summarize the main parameters of the model. A dagger (†) in the last column indicates that example values could be taken or estimated from empirical data.

C.4   Simplifying Assumptions

Like every simulation model, this simulation model is built for a specific purpose (comparison of pre- and post-commit reviews) and contains simplifying assumptions, notably:

- A developer works on only one task at a time.
- Breaks and interruptions are not modeled, as are weekends and breaks between working days.
- There is a strict priority order of task types.
- The review round does not influence the time for a review, as well as the probability of finding an issue.
- A review is never canceled, even if there is a very high number of issues found.
- Review effects other than defect detection and spent effort (e. g. better knowledge distribution or social problems) are not modeled. Neither are psychological effects.
- The model representation of the planning process is very simplistic (a developer starts planning alone, other developers join anytime, . . .).
- While working on a task, a developer will not update his working copy.
- The process of developers finding issues while working on other tasks can be modeled with an exponential distribution.
- Soon after they are finished, stories will reach the customer (like continuous delivery with cherry picking) without further issues being injected (unlike cherry picking).
- The model contains no distinction between issues found by customers and issues found by system testing after development. That is, 'issues found by customers' really means 'issues found by customers or by external QA' in the context of this study.
- The "topic" that determines if a change of topic leads to task switch overhead is the story the task belongs to.
- Dependencies between tasks only occur within a story.
- The same distribution is used for the issue assessment times of issues found by developers and issues found by customers.
- Normal issues can only be found once by a developer and once by a customer.

Table C.1: Main model parameters (1/2). The example values marked with † are derived from the partner company's ticket system.

Name/topic	Description	Sampling range (Sec. 10.2.2)	Example value (Sec. 10.2.1)
Issue injection rate	The mode and width of a triangular distribution giving the implementation skills of the developers, measured in issues injected per implementation task hour.	0.0 – 1.5 i./h 0.0 – 0.4 i./h	0.26 i./h† 0.1 i./h
Review effectiveness	The mode and width of a triangular distribution giving the code reviewing skills of the developers, measured as the probability of detecting an issue in review.	0.0 – 1.0 0.0 – 0.3	0.45 0.05
Global blocker issue risk	The probability that a global blocker issue is injected while implementing a task or fix.	0.0 – 0.1	0.001
Global blocker issue suspend time	Mode of a triangular distribution taken for sampling the time for interruptions through global blocker issues.	0.01 – 3.0 h	0.15 h
Conflict probability	The probability that a conflict occurs between a task and another specific task that was committed between the first task's update and now.	0.0 – 0.1	0.012
Conflict resolution time	Mode of a triangular distribution taken for sampling the time for resolving a conflict on commit.	0.1 – 3.0 h	0.3 h
Implementation task duration	Mode and difference between mode and mean for the distribution from which a story task's duration (without overhead) is sampled. A log-normal distribution provided the best fit to the empirical data, a shifted exponential distribution was used as a second alternative (this applies to all durations).	0.1 – 2.0 h 0.1 – 28.0 h	0.2 h† 9.5 h†
Review duration	Mode and difference between mode and mean for the distribution from which the time needed for a single review round is sampled.	0.02 – 1.0 h 0.1 – 6.0 h	0.0323 h† 1.625 h†
Review remark fix duration	Mode and difference between mode and mean for the distribution from which the time needed to fix a single review remark is sampled.	0.01 – 0.25 h 0.01 – 4.0 h	0.02 h 0.7 h
Issue fix task overhead	Mode and difference between mode and mean for the distribution from which the overhead analysis time for fixing an issue as an issue fix task (in addition to the time needed to fix it as review remark) is sampled.	0.01 – 2.0 h 0.1 – 28.0 h	0.0132 h† 5.5 h†
Issue assessment duration	Mode and difference between mode and mean for the distribution from which the time needed for issue assessment is sampled.	0.01 – 1.0 h 0.1 – 4.0 h	0.1 h 0.35 h
Internal issue share	Share of issues that can not be found by customers.	0.0 – 1.0	0.5
Issue activation time – Developer	Shift and mean for a shifted exponential distribution giving the time between committing an issue and its surfacing to a developer.	0.0 – 8.0 h 4 - 4000 h	0.5 h 2000 h
Issue activation time – Customer	Shift and mean for a shifted exponential distribution giving the time between "delivering" an issue and its surfacing to a customer.	4.0 h 4 - 4000 h	4.0 h 1000 h†

Table C.2: Main model parameters (2/2)

Name/topic	Description	Sampling range (Sec. 10.2.2)	Example value (Sec. 10.2.1)
Planning duration	Mean of the distribution from which the time needed for planning a story is sampled.	0.1 − 30.0 h	4.0 h
Developer count	Number of developers working in the team/same part of the software.	3 − 29	12[†]
Task switch overhead	Time it takes for a developer to re-familiarize himself with a topic after working on another topic for (1) one hour (2) a very long time.	0.01 − 0.5 h 0.5 − 3.0 h	5 min. 30 min.
Task switch issue factor	Factor that determines to which amount time needed for task switch will be taken into account when calculating the number of issues injected.	0.0 − 1.0	0.0
Fixing issue rate factor	Factor for the issue injection rate that determines how safe fixing is compared to creating new code.	0.2 − 1.1	0.3
Follow up issue spawn probability	Probability that an issue in a predecessor task leads to a follow up issue in a dependent task.	0.0 − 0.15	0.005
Review fix to task factor	Factor determining how time-consuming fixing an issue (without finding it) is in an issue fix task compared to a review remark.	0.9 − 2.0	1.1
Dependency graph constellation	Different types of task dependency structures, for example "NO_DEPENDENCIES" when there are no (relevant) dependencies between tasks of a story and "REALISTIC" for empirically observed structures.	NO_DEP./ REALIS- TIC	REALISTIC[†]
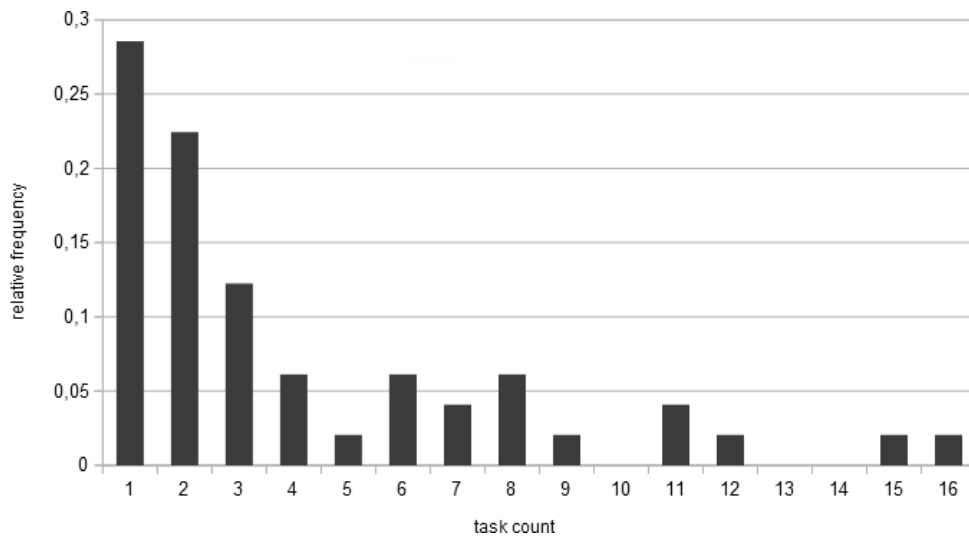
Figure C.8: Relative frequency of different task counts per user story in the empirical sample

D

An Efficient Algorithm to Find an Optimally Ordered Tour

Section 14.5 formalized what is meant by an optimal tour. The current chapter builds upon this formalization and presents a polynomial time algorithm to find an optimal tour, given a collection of pattern matches.

D.1 Description of the Algorithm

This section outlines the algorithm. In many cases, there are several mutually incomparable optimal tours, because $\geq_T$ is only a partial order. Putting further constraints on which of these tours to select can increase the computational complexity of the problem. For example, the problem becomes NP-hard when looking for an optimal tour with a maximal number of non-clustered pattern matches (proof by reduction from Hamiltonian path). The algorithm presented here takes a different route: It assumes a list of pattern matches as input that is ordered from most to least important. The optimal tour is then found greedily, trying to satisfy the most important pattern match first.

Another clarification is needed for the function $rate$ that was left unspecified in Section 14.5. Both the results from this chapter's survey (Table 14.3) and from Fregnan's thesis [129] indicate a preference for starting with the center of a star pattern. The respective rate function is:

$$rate_{bottomUp}(m, t) = \begin{cases} 1 & \text{if the center of } m \text{ is before all other change parts of } m.v \text{ in } t, \\ 0 & \text{otherwise} \end{cases}$$

The notion of shrinking the part graph, as used in the formalization of the theory in Section 14.5, is intuitive for humans, but it can be further simplified to ease the construction of the algorithm. The simplification is based on the observation that the only information needed from an undirected pattern match is the set of matched change parts. And for directed matches, with the restriction to the 'bottomUp' rate function, it is sufficient to have two sets for 'center' and 'rest'. Instead of looking for matches in a shrunk graph, the matches from the original sets can also be 'expanded' before checking whether they are satisfied for a tour.

To ease the formulation of the algorithm, this thesis proposes a novel abstract data type called 'Binder' (B). The intuition is that such a binder can be used to accumulate requests to 'bind together' a group of change parts, as long as they do not conflict with earlier requests, and finally to get a tour that satisfies the accumulated requests. Its interface is as follows:

$$\text{create} : \mathcal{P}\left(ChangePart\right) \to B$$

$$\text{bind} : B \times \mathcal{P}\left(ChangePart\right) \to B$$

$$\text{bindOrdered} : B \times \mathcal{P}\left(ChangePart\right) \times \mathcal{P}\left(ChangePart\right) \to B$$

$$\text{get} : B \to Tour$$

For ease of specification, a binder $b \in B$ is regarded as a triple (e, s, o), with e the set of change parts that need to be ordered, s the set of satisfiable matches added so far, and o the additional center/rest information for the subset of s that are ordered matches.

$$\text{create}(e) := (e, \emptyset, \emptyset)$$

$$\text{bind}((e, s, o), m) := \text{addIfSatisfiable}\left(e, s, \{m\}, o, \emptyset\right)$$

$$\text{bindOrdered}((e, s, o), m_c, m_r) := \text{addIfSatisfiable}\left(e, s, \{m_c \cup m_r\}, o, \{(m_c, m_r)\}\right)$$

$$\text{get}((e, s, o)) := t \text{ such that } \text{satisfiesAll}(t, e, s, o)$$

$$\text{addIfSatisfiable}(e, s_1, s_2, o_1, o_2) := \begin{cases} (e, s_1 \cup s_2, o_1 \cup o_2) & \text{if } \exists t : \text{satisfiesAll}(t, e, s_1 \cup s_2, o_1 \cup o_2) \\ (e, s_1, o_1) & \text{otherwise} \end{cases}$$

$$\begin{aligned} \text{satisfiesAll}(t, e, s, o) := {} & t \text{ is a permutation of } e \\ & \wedge\ \forall m \in s : \text{satisfiesBind}(t, m) \\ & \wedge\ \forall (m_c, m_r) \in o : \text{satisfiesOrder}(t, m_c, m_r) \end{aligned}$$

$$\text{satisfiesBind}(t, m) := \exists i \in \mathbb{N} : \forall j \geq i, j < i + |m| : t_j \in m$$

$$\text{satisfiesOrder}(t, m_c, m_r) := \forall i, j \in \mathbb{N} : t_i \in m_c \wedge t_j \in m_r \Rightarrow i < j$$

The wording "A call to 'bind' or 'bindOrdered' could be satisfied" is used for the situation where the call led to a Binder which reflects the parameters in the call.

The description of the algorithm itself is split over the Figures D.1, D.2, and D.3. The algorithm has three main phases. In the first, it tries to satisfy as many match sets as possible without taking folding into account. In the second, it tries to satisfy further match sets with folding. In the final phase, it tries to satisfy the bottom-up ordering for ordered match sets induced by $rate_{bottomUp}$. The first and third phase are simple greedy loops, the second phase is more complicated. It needs to find a minimal set of folds that satisfy further match sets. Simply testing all subsets would not be possible in polynomial time. Instead, it starts with applying all folds at once and then leaves out unnecessary folds until a minimal set is found. The algorithm's correctness is proven in Section D.3.

A rough estimation of the runtime complexity of the presented algorithm shows that is $O(m^4 \cdot b(m, n))$, with m being the number of match sets, n the number of change parts and b a

placeholder for the runtime complexity of the *bind* operation. Most of the runtime complexity lies in the handling of folds. Section D.2 introduces an implementation of the abstract data type Binder for which b is $O(n)$, as are the other operations. In the worst case, the number of match sets m is $O(n^2)$. All combined, this leads to an upper bound of the worst-case complexity of $O(n^9)$, which is better than the exponential complexity that would follow from the direct implementation of the definitions in Section 14.5. The actual implementation of the algorithm in CoRT contains some further optimizations to avoid testing unnecessary folds.

Often, there are far fewer than n^2 match sets, and many of them can be satisfied without folding. In these cases, the performance of the algorithm is good enough for use in industrial practice. But cases with a high number of change parts or match sets occur in practice. One example are large systematic changes, for which many pairs of change parts are related by similarity. These cases led to problems in an early implementation of the algorithm in CoRT. In these cases, CoRT takes advantage of the incremental nature of the algorithm. When the calculation is stopped prematurely, the intermediate state of the binder contains the successfully satisfied match sets. So CoRT monitors the time spent for finding the optimal tour. If the user has to wait for too long, the optimization is stopped prematurely and the current intermediate result is used.

```
function determineOptimalOrder(changeParts, matchSets)
  binder = Binder.create(changeParts)
  toSatisfy = matchSets
  satisfied = empty set

  //Phase 1: bind everything that can be bound without folding/expanding
  (binder, newlySatisfiedMatches) = bindAll(binder, toSatisfy)
  toSatisfy.removeAll(newlySatisfiedMatches)
  satisfied.addAll(newlySatisfiedMatches)

  //Phase 2: take folding into account
  loop
    //expand the unsatisfied matches (equivalent to folding the tour)
    //  and try to satisfy further matches with a minimal set of folds
    toFold = findMinimalSetOfSuccessfulFolds(binder, toSatisfy, satisfied)
    if (did not find a successful fold)
      //no further match sets can be satisfied, even with folding
      break
    else
      toSatisfy = expandAll(toSatisfy, toFold)
      (binder, newlySatisfiedMatches) = bindAll(binder, toSatisfy)
      toSatisfy.removeAll(newlySatisfiedMatches)
      satisfied.addAll(newlySatisfiedMatches)
    end-if
  end-loop

  //Phase 3: try to satisfy the ordering requests
  foreach matchSet in satisfied
    if (matchSet demands bottom-up order)
      binder = binder.bindOrdered(matchSet.center, matchSet.rest)
    end-if
  end-foreach

  return binder.get()
end-function
```

Figure D.1: Pseudo-code description of the algorithm to determine an optimal tour (1/3): Main algorithm. This is a simplified version, the full algorithm contains additional performance optimizations and is distributed with CoRT.

```
function findMinimalSetOfSucessfulFolds(binder, toSatisfy, foldsToTry)
  //start with applying all folds and then remove folds
  //   as long as matches can still be satisfied
  if (!allowsFurtherBinds(binder, foldsToTry))
    return empty set
  end-if
  do
    unnecessaryFold = findFoldThatCanBeLeftOut(binder, toSatisfy,
        foldsToTry)
    foldsToTry.remove(unnecessaryFold)
  while (unnecessaryFold != null)
  return foldsToTry
end-function


function findFoldThatCanBeLeftOut(binder, toSatisfy, foldsToTry)
  foreach fold in foldsToTry
    if (allowsFurtherBinds(binder, toSatisfy, foldsToTry / {fold}))
      return fold
    end-if
  end-foreach
  return null
end-function


function allowsFurtherBinds(binder, toSatisfy, foldsToApply)
  toSatisfy = expandAll(toSatisfy, foldsToApply)
  ( , newlySatisfiedMatches) = bindAll(binder, toSatisfy)
  return !newlySatisfiedMatches.isEmpty()
end-function
```

Figure D.2: Pseudo-code description of the algorithm to determine an optimal tour (2/3): Finding the folds to apply.

```
function bindAll(binder, toBind)
  newlySatisfiedMatches = empty set
  foreach matchSet in toBind
    binder = binder.bind(matchSet)
    if (bind was possible)
      newlySatisfiedMatches.add(matchSet)
    end-if
  end-foreach
  return (binder, newlySatisfiedMatches)
end-function

function expandAll(toSatisfy, foldsToApply)
  expandedMatchSets = empty list
  foreach m in toSatisfy
    expandedMatchSets.add(expand(m, foldsToApply))
  end-foreach
  return expandedMatchSets
end-function

function expand(m, foldsToApply)
  expanded = m
  foreach n in foldsToApply
    if (m and n are not disjoint)
      expanded.addAll(n)
    end-if
  end-foreach
  return expanded
end-function
```

Figure D.3: Pseudo-code description of the algorithm to determine an optimal tour (3/3): Utility functions.

D.2 An Implementation of the Abstract Data Type 'Binder'

This section illustrates a tree-based implementation of the abstract data type 'Binder', which was specified in the previous section. It focuses on the idea and intuition. The full implementation is complicated by the handling of several special cases and can be seen in the implementation of CoRT (component "ordering"; see Section 9.4).

A 'BinderTree' is a tree in which every leaf node corresponds to exactly one change part and which has exactly one leaf node for every change part. The inner nodes can be of several types, and each type restricts the possible permutations of the children in a specific way. The possible types for inner nodes are:

- SET: Denotes that every permutation of the children leads to a valid order.
- SEQUENCE: Denotes that the children must be sorted in their given order or reversed in total to give a valid order.
- FIXED: Denotes that the children must be sorted in their given order and may not even be reversed.

Semantically, a BinderTree stands for a subset of the permutations of the changesets. The semantic for a leaf node is the one-element sequence with the change part itself. The semantic function s for the intermediate nodes is as follows:

$$s : BinderTree \rightarrow \mathcal{P}(Tour)$$

$$s(set(c_1, c_2, \ldots, c_n)) := \bigcup_{p \in S_n} s(c_{p(1)}) \times s(c_{p(2)}) \times \cdots \times s(c_{p(n)})$$

$$s(sequence(c_1, c_2, \ldots, c_n)) := (s(c_1) \times s(c_2) \times \cdots \times s(c_n)) \cup (s(c_n) \times \cdots \times s(c_2) \times s(c_1))$$

$$s(fixed(c_1, c_2, \ldots, c_n)) := s(c_1) \times s(c_2) \times \cdots \times s(c_n)$$

Here, S_n denotes the set of all permutation operations on n items.

In the following, the operations on a BinderTree are illustrated by various examples. The examples use a short-hand notation for BinderTrees in which $set(\ldots)$ stand for a SET node, $seq(\ldots)$ stands for a SEQUENCE node, $fix(\ldots)$ stands for a FIXED node, and leaf nodes are denoted by single uppercase letters. Figure D.4 depicts this notation.

Assume that there are eight change parts, A to H. The initial BinderTree does not restrict the possible permutations at all:

$$create(\{A, B, C, D, E, F, G, H\}) = set(ABCDEFGH)$$

Binding together certain change parts results in a tree that reflects the reduced set of possible permutations:

$$bind(set(ABCDEFGH), \{A, B, C, D\}) = set(set(ABCD)EFGH)$$
$$bind(set(set(ABCD)EFGH), \{E, F, G, H\}) = set(set(ABCD)set(EFGH))$$

set(seq(ABC)DEset(FGH))
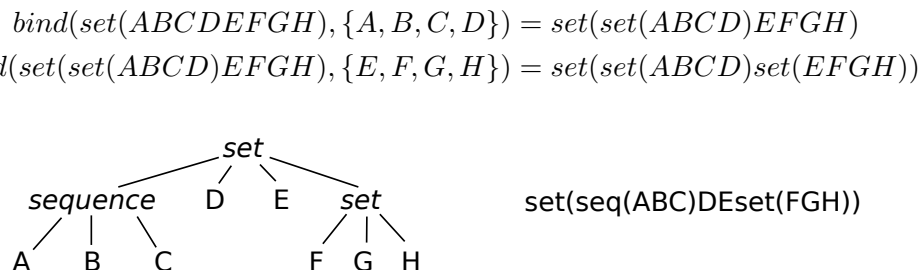
Figure D.4: Example of a BinderTree and its textual notation

When change parts that reside in different subtrees shall be bound together, sequence nodes need to be introduced so that the parts cannot be torn apart any more:

$$bind(set(set(ABCD)set(EFGH)), \{A, H\}) = seq(set(BCD)AHset(EFG))$$
$$bind(seq(set(BCD)AHset(EFG)), \{A, B, C\}) = seq(Dset(BC)AHset(EFG))$$
$$bind(seq(Dset(BC)AHset(EFG)), \{C, D\}) = seq(DCBAHset(EFG))$$

Sometimes, a sequence node needs to be reversed to allow satisfying a bind:

$$bind(set(ABCDEFGH), \{A, B\}) = set(set(AB)CDEFGH)$$
$$bind(set(set(AB)CDEFGH), \{B, C\}) = set(seq(ABC)DEFGH)$$
$$bind(set(set(ABC)DEFGH), \{D, E\}) = set(seq(ABC)set(DE)FGH)$$
$$bind(set(seq(ABC)set(DE)FGH), \{E, F\}) = set(seq(ABC)seq(DEF)GH)$$
$$bind(set(seq(ABC)seq(DEF)GH), \{C, F\}) = set(seq(ABCFED)GH)$$

Sequence nodes can occur inside sequence nodes, as set nodes can occur inside set nodes:

$$bind(set(set(seq(ABC)seq(DEF))seq(GHI)), \{D, E, F, G, H, I\}) =$$
$$seq(seq(ABC)seq(DEF)seq(GHI))$$

Because the in-order traversal of the tree leaves always leads to a valid permutation, the get function simply needs to return this order:

$$get(set(seq(ABCFED)GH)) = (A, B, C, F, E, D, G, H)$$

The 'bindOrdered' operation binds together the center set, the rest set, and the union of center and rest. Furthermore, it needs to assure that these will not be reversed anymore by introducing a FIXED node:

$$bindOrdered(set(ABCDEFGH), \{A, B, C\}, \{D, E, F\}) = set(fix(set(ABC)set(DEF))GH)$$

D.3 Proof of Correctness for the Ordering Algorithm

Section 14.5 defined the partial order relation $\geq_T$ and Section D.1 introduced an algorithm to determine a maximal element regarding $\geq_T$. This section now proves that this algorithm works correctly.

The general structure is that of a proof by contradiction, showing that assuming both that t_a is the result of the algorithm and that there is an element $t_b >_T t_a$ leads to a contradiction. Like the algorithm, the proof does not use the graph-based formulation of the theory but is instead based on sets of change parts that need to be grouped. Before showing the proof itself, this alternative definition of $>_T$ is given.

Definition 4 (MatchSet). A match set is a set of change parts that are matched by a rule. The set "MatchSet" is the set of all possible match sets:

$$\text{MatchSet} := \mathcal{P}\,(\text{ChangePart})$$

For ordered matches for a star pattern, it is also assumed that there are projections 'center' and 'rest' that return the respective disjoint subsets of change parts for the match set.

Definition 5 (SatisfiedMatch). 'SatisfiedMatch' is a shorthand notation for a set of pairs of a match set and a set of match sets that need to be considered in shrinking the graph to satisfy it for a given tour:

$$\text{SatisfiedMatch} := \text{MatchSet} \times \mathcal{P}\,(\text{MatchSet})$$

Definition 6 (sM). The function 'sM' from Section 14.5 can be redefined based on match sets:

$$\text{sM} : \text{Tour} \times \mathcal{P}\,(\text{MatchSet}) \to \mathcal{P}\,(\text{SatisfiedMatch})$$

$$\text{sM}(t, M) := \text{sME}(t, M, \emptyset)$$

The second parameter to 'sM' are the sets of change parts that were matched in the change part graph. Therefore, the graph and the set of patterns can be left out from this definition. The recursive helper function 'sME' takes the match sets that shall be considered for expansion/shrinking as the third parameter:

$$\text{sME} : \text{Tour} \times \mathcal{P}\,(\text{MatchSet}) \times \mathcal{P}\,(\text{MatchSet}) \to \mathcal{P}\,(\text{SatisfiedMatch})$$

$$\text{sME}(t, M, E) := \text{pM}(t, M, E) \cup \bigcup_{(m,\_)\in\text{pM}(t,M,E)} \text{sME}(t, M, E \cup \{m\})$$

The helper function 'pM' returns the match sets that are satisfied for a given tour and set of expansions (with 'satisfiesBind' as defined in Section D.1):

$$\text{pM} : \text{Tour} \times \mathcal{P}\,(\text{MatchSet}) \times \mathcal{P}\,(\text{MatchSet}) \to \mathcal{P}\,(\text{SatisfiedMatch})$$

$$\text{pM}(t, M, E) := \{(m, E) : m \in M \land satisfiesBind(t, expand(m, E))\}$$

Definition 7 (expand). The function 'expand' enlarges a match set by joining it with overlapping match sets:

$$expand : \text{MatchSet} \times \mathcal{P}(\text{MatchSet}) \rightarrow \text{MatchSet}$$

$$expand(m, E) := \begin{cases} expand(m \cup x, E \setminus \{x\}) & \exists x \in E : m \cap x \neq \emptyset \\ m & \text{otherwise} \end{cases}$$

For ordered matches of a star pattern, expanding the center is preferred:

$$expand(m, E).center = expand(m.center, E)$$

Definition 8 ($>_T$). Based on the original definition of $>_T$ and $\geq_T$ as given in Section 14.5, an alternative definition of $>_T$ can be given as:

$$t_1 >_T t_2 \iff \text{sM}(t_1, M) \supset \text{sM}(t_2, M) \vee$$
$$(\text{sM}(t_1, M) = \text{sM}(t_2, M)$$
$$\wedge \forall (m, E) \in \text{sM}(t_1, M) : \text{rate}(expand(m, E), t_1) \geq \text{rate}(expand(m, E), t_2)$$
$$\wedge \exists (m, E) \in \text{sM}(t_1, M) : \text{rate}(expand(m, E), t_1) > \text{rate}(expand(m, E), t_2))$$

It is parametric, based on the set M of MatchSets derived from the pattern matching in the part graph. In the following, $\text{rE}(m, E, t)$ is used as a shorthand for 'rate(expand(m, E), t)'.

With these preparations in place, this chapter's main proposition can be stated:

Proposition D.1 (Correctness of ordering algorithm). For a given set C of change parts and $M \subseteq \text{MatchSet}$ (with $\text{MatchSet} = \mathcal{P}(C)$) and any permutation M' of M, it holds that $t_a := \text{determineOptimalOrder}(C, M')$ is a maximal element of $\geq_T$, i.e., there is no t_b such that $t_b >_T t_a$. The implementation of 'determineOptimalOrder' is given in Figure D.1 in Section D.1.

Proof of Proposition D.1, High-level overview. Assume that $t_a := \text{determineOptimalOrder}(C, M')$ and that there exists an $t_b >_T t_a$. According to Definition 8, two cases can be distinguished:

Case 'set difference': $\text{sM}(t_b, M) \supset \text{sM}(t_a, M)$

Case 'rating difference': $\text{sM}(t_b, M) = \text{sM}(t_a, M) \wedge \forall (m, E) \in \text{sM}(t_b, M) : \text{rE}(m, E, t_b) \geq \text{rE}(m, E, t_a) \wedge \exists (m, E) \in \text{sM}(t_b, M) : \text{rE}(m, E, t_b) > \text{rE}(m, E, t_a)$

As is shown later, both cases lead to contradictions. Therefore, such a t_b cannot exist and t_a must be maximal. $\square$

To prove Case 'set difference', some auxiliary definitions are useful:

Definition 9 (expand path). An 'expand path' of length n is a pair $(binds, expands)$ with

$$binds := bind_0, bind_1, \ldots, bind_n$$

$$bind_i \subseteq \text{MatchSet}$$

$$expands := expands_1, \ldots, expands_n$$

$$\emptyset \subset expands_i \subseteq \bigcup_{0 \leq j < i} bind_j$$

It is useful to have a short-hand notation for the union of expanded match sets in the path up to length i:

$$ems(p, i) := \{expand(m, \bigcup_{1 \leq k \leq j} p.expands_k) : j \leq i \land m \in p.bind_j\}$$

The intuition behind an expand path is that the match sets in $bind_0$ can be satisfied without expanding, those in $bind_1$ can be satisfied by applying the expansions in $expands_1$, those in $bind_2$ by applying the expansions in $expands_1$ and $expands_2$, and so on.

Definition 10 (expand path for algorithm call). Each call to 'determineOptimalOrder' corresponds to an expand path as follows:
- $bind_0$ is the set of satisfied matches after Phase 1
- $expands_i$ corresponds to the value of 'toFold' after the i-th iteration of Phase 2's loop
- $bind_i, i \geq 1$ corresponds to the unexpanded versions of the newly satisfied match sets after the i-th iteration of Phase 2's loop

Definition 11 (conflict). The predicate 'conflict' is true for a set of match sets when the match sets cannot be satisfied jointly:

$$conflict(M) := \nexists t \in Tour : satisfiesAll(t, C, M, \{m \in M : m \text{ is ordered match}\})$$

with $satisfiesAll$ as defined in Section D.1, and C the set of change parts. Besides the 'one set' variant of the predicate defined above, a variant that takes two arguments is useful, too: $conflict(m, M) := conflict(\{m\} \cup M)$.

As the algorithm heavily relies on the abstract data type Binder, the following Lemma is useful later:

Lemma D.1. Every call to 'bind' or 'bindOrdered' for a match set m and a Binder B can either be satisfied or there is a non-empty set S of match sets used in earlier calls that cannot be satisfied jointly with m.

Proof of Lemma D.1. The proof is done by structural induction over the interface of Binder.
Induction start, operation 'create': $B = create(s)$. For the newly created binder, every call to 'bind' or 'bindOrdered' can be satisfied so that the Lemma is trivially true.
Induction step, operation 'bind': $B = bind(B', x)$. When $bind(B', m)$ cannot be satisfied, then

there exists a non-empty set S' that cannot be satisfied jointly with m by the induction pre-condition. Adding further calls to 'bind' cannot make non-satisfiable match sets satisfiable, so that $bind(B, m)$ cannot be satisfied, too and $S \supseteq S'$ is non-empty. When $bind(B', m)$ can be satisfied but $bind(B, m)$ cannot, x must be an element of S and S is non-empty.

Induction step, operation 'bindOrdered': Like for 'bind'. □

The expand path for a run of the algorithm as specified in Definition 10 satisfies two properties that are needed later:

Lemma D.2. Let p be the expand path for the call determineOptimalOrder(C, M). For all $bind_i$ in p, it holds that

$$\forall m \in M, m \notin \bigcup_{0 \leq k \leq i} bind_k : conflict(expand(m, \bigcup_{1 \leq l \leq i} expands_l), ems(p, i))$$

I.e., the $bind_i$ are complete in a way that no further match set could be added without conflicting with another one.

Proof for Lemma D.2. Case $i = 0$: $bind_0$ corresponds to the matches that could be satisfied in Phase 1. In Phase 1, the algorithm calls 'bindAll' to try to bind all match sets in M. If such a call to 'bind' cannot be satisfied, there exists, according to Lemma D.1, another match set m' that was satisfied earlier. Therefore, m' is contained in $bind_0$.

Case $i > 0$: The items in $bind_i$ for $i > 0$ are determined by calling 'bindAll' in the i-th iteration of Phase 2's loop. The binds that are tried are obtained by expanding the original match sets with the items in $expands_1$ to $expands_i$. The same binder is kept from one iteration to the next. Again, if a call to 'bind' cannot be satisfied there must be another match set m' that was satisfied earlier (Lemma D.1). □

Lemma D.3. Let p be the expand path for the call determineOptimalOrder(C, M). For all $expands_i$ in p, it holds that

$$\forall e \in expands_i : \exists m \in bind_i, m_e = expand(m, \bigcup_{1 \leq j \leq i} expands_j \setminus \{e\}) :$$

$$conflict(m_e, ems(p, i))$$

I.e., the $expands_i$ are minimal in a way that removing one will make at least one match set in the path unsatisfiable jointly with the rest.

Proof of Lemma D.3. The set $expands_i$ is determined by a call to 'findMinimalSetOfSuccessful-Folds'. The loop in that operation only stops when any further removal of an element would make any additional bind impossible. □

Proof of Proposition D.1, Case 'set difference'. It needs to be shown that the assumption that $sM(t_b, M) \supset sM(t_a, M)$ when t_a is the result of the ordering algorithm leads to a contradiction. Spelled out, the assumption can be written as $\exists x \in sM(t_b, M) : x \notin sM(t_a, M) \land \forall y \in sM(t_a, M) : y \in sM(t_b, M)$. According to the definition of sM, x can be written as a pair (m, E), meaning that m can be satisfied after the expansions in E are applied.

The proof is done by induction over the size of E.

Induction start: $E = \emptyset$. This means that m can be satisfied without expansion. According to Definition 10, there is an expand path p corresponding to t_a, and m would be in $bind_0$ of that path. According to Lemma D.2, there is an element c in $bind_0$ which cannot be satisfied jointly with m. Therefore, $c \in \mathrm{sM}(t_a, M)$ and $c \notin \mathrm{sM}(t_b, M)$, which contradicts this case's precondition. Such a $(m, \emptyset)$ cannot exist.

Induction step: To show: Assuming that $\nexists (m, E) \in \mathrm{sM}(t_b, M) : |E| < n \wedge (m, E) \notin \mathrm{sM}(t_a, M)$, there is also no $(m, E) \in \mathrm{sM}(t_b, M) : |E| = n \wedge (m, E) \notin \mathrm{sM}(t_a, M)$. This is shown by separating a number of cases:

- Case 1: $\exists E' : E' \subset E \wedge (m, E') \in \mathrm{sM}(t_b, M)$. By the induction precondition, it follows that $(m, E') \in \mathrm{sM}(t_a, M)$. By adding further expansions, a satisfied match cannot be made unsatisfied, so $(m, E) \in \mathrm{sM}(t_a, M)$, leading to a contradiction.
- Case 2: $\nexists E' : E' \subset E \wedge (m, E') \in \mathrm{sM}(t_b, M)$. Again, p is the expand path for t_a. Determine the smallest i such that $\bigcup_{1 \leq j \leq i} p.expands_j = U \supseteq E$.
  - Case 2.1: There is no such i. This can only happen when one of the $e \in E$ could not be satisfied, i.e., $\exists (e, E'') \in \mathrm{sM}(t_b, M) : |E''| < |E| \wedge (e, E'') \notin \mathrm{sM}(t_a, M)$. This contradicts the induction precondition.
  - Case 2.2: There is such a i.
    * Case 2.2.1: $U = E$. Similar to the induction start, it can be shown based on Lemma D.2 that this leads to a contradiction.
    * Case 2.2.2: $U \supset E$. The following again uses p as notation for the expand path for t_a.
      · Case 2.2.2.1: $m \in p.bind_i$. According to the condition for Case 2.2.2, there is an $x \in U, x \notin E$ that could be removed from the expand path without affecting the satisfiability of m in $p.bind_i$. But according to Lemma D.3, there is another $(m', E^*) \in \mathrm{sM}(t_a, M)$ that depends on this x. This contradicts the precondition that $\forall e \in \mathrm{sM}(t_a, M) : e \in \mathrm{sM}(t_b, M)$
      · Case 2.2.2.2: $m \in p.bind_j, j < i$. In this case, there must be an $(m, E^*) \in \mathrm{sM}(t_a, M)$ with $E^* \subset U$ but not $E^* \subset E$. Respectively, there is an $x \in E^*, x \notin E$. Like for Case 2.2.2.1, it can be argued based on Lemma D.3 that this leads to a contradiction.
      · Case 2.2.2.3: $m \notin \bigcup_{0 \leq j \leq i} p.bind_j$. According to Lemma D.2, there must be another match set that conflicts with (m, U). When (m, U) cannot be satisfied, (m, E) cannot be satisfied, either (contraposition of the observation that adding items to the expands set will never lead to conflicts).

As all possible cases lead to contradictions, it can be concluded that $\mathrm{sM}(t_b, M)$ is not a superset of $\mathrm{sM}(t_a, M)$. □

Proof of Proposition D.1, Case 'rating difference'. With this case's precondition, $\mathrm{sM}(t_b, M) = \mathrm{sM}(t_a, M)$. Note that $\mathrm{sM}(t_a, M)$ is reflected in the variable 'satisfied' in the function 'determineOptimalOrder'.

With the 'bottom-up' definition of 'rate' from Section D.1,

$$\exists (m, E) \in \mathrm{sM}(t_b, M) : \mathrm{rE}(m, E, t_b) > \mathrm{rE}(m, E, t_a)$$

can be stated as

$$\exists (m, E) \in \mathrm{sM}(t_b, M), m_e = \mathrm{expand}(m, E) :$$
$$satisfiesOrder(t_b, \mathrm{center}(m_e), \mathrm{rest}(m_e)) \wedge \neg satisfiesOrder(t_a, \mathrm{center}(m_e), \mathrm{rest}(m_e))$$

It follows from the initial observation that every element in $\text{sM}(t_b, M)$ was tried to satisfy by calling 'bindOrdered' in Phase 3 of the algorithm. So 'bindOrdered' was also called for m_e. There are two possible outcomes: The call to 'bindOrdered' was successful for m_e. In this case, $satisfiesOrder(t_a, \text{center}(m_e), \text{rest}(m_e))$ must have been true, leading to a contradiction. Or the call to 'bindOrdered' was not successful for m_e. In this case, an earlier call to 'bind' or 'bindOrdered' must have rendered it impossible (Lemma D.1). It cannot have been a call to 'bind', because $\text{sM}(t_b, M) = \text{sM}(t_a, M)$ and all successful binds are reflected in the 'satisfied' variable. According to Lemma D.1, there must be a non-empty set $S \subset \text{sM}(t_b, M)$ which cannot be satisfied together with m_e and which is satisfied for t_a. For the elements in $s \in S$, $\text{rE}(s, E, t_b) = 0 < \text{rE}(s, E, t_a) = 1$, which again contradicts the for-all-clause in the case's precondition. $\qquad\square$

$$\Diamond$$

All in all, Proposition D.1 could be proven. This shows that the ordering algorithm will find an order of change parts that is a maximal element according to $\geq_T$ and, therefore, a good order for review. The definition of $\geq_T$ allows for various mutually incomparable maximal elements, so this result does not mean that the algorithm will return *the* single empirically best order (if such an order should exist at all).

E

Details on How to Extract Review Remark Triggers

The study on change part classification in Chapter 15 needs data on which change parts can act as triggers for which review remarks. The following sections provide details on how this information can be extracted from the SCM and ticket system at the partner company. The final Section E.5 furthermore provides evidence that it is not sufficient to use the simpler SZZ approach [353] to trace review remarks to their triggers.

E.1 Remarks, Triggers, and Change Parts

To judge the importance of a change part for review, one needs to tell whether it contains triggers for review remarks. A trigger is the portion of code whose review leads to the creation of a review remark. In a simple case, the trigger is a defect in a specific line of the code. Another example could be a misspelled or confusing method name: Every occurrence of the method name in the code can lead to the observation of the problem and creation of the remark. As this example shows, the relation between remarks, triggers, and change parts is not a simple one-to-one relation: There can be multiple types of composite conditions. Several of the possibilities are depicted in Figure E.1.

The simplest possibility is when a change part acts as a trigger for exactly one review remark, as for C1 and R1. A single change part can also lead to several remarks (C2, R2, and R3). In some cases, there can be more than one trigger for a remark. One example is the typo in a method name mentioned above. This situation is similar to a logical 'or': R4 is triggered when C3 or C4 is reviewed. In other cases, the knowledge of several parts of the code is needed to spot a problem and trigger a remark, i.e., a logical 'and'. The figure shows a combined case, where reviewing C5 and at least one of C6 or C7 triggers the remark R5. There are also change parts that do not act as a trigger at all (C8), which is the primary motivation for the current study. The concept of "and" relations in triggers is related to the issue of understandability for the reviewer mentioned in the previous section. It is not studied further in this thesis.
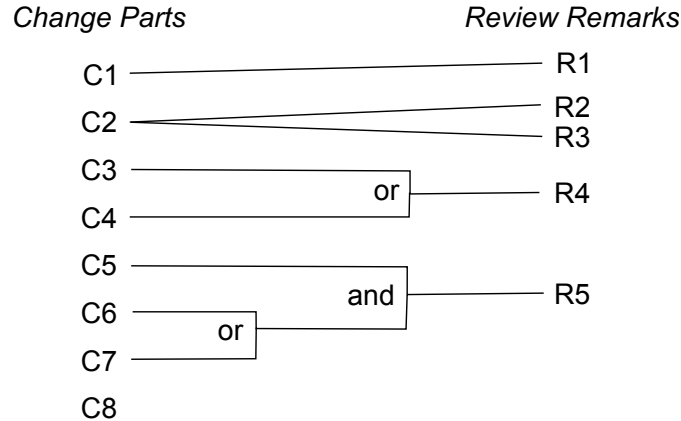
Figure E.1: Several possibilities how change parts can act as triggers for review remarks

E.2 Selecting a Data Source

This section discusses how the information on potential review triggers can be extracted from software repositories. The technique is based on the assumption that one of the triggers for a review remark is close to the position of the remark.

There are two possibilities to gather review remarks from software repositories: (1) Communicated review remarks can be extracted from the repository of the code review tool; or (2) fixed review remarks can be extracted from the SCM. Although there usually is a high overlap between communicated and fixed remarks, there is a subset of non-fixed remarks that cannot be found in the SCM and there are remarks that the reviewer fixes 'on-the-fly' without recording them in the review discussion[1]. Table E.1 contains a detailed discussion of benefits of both approaches to extract review remarks. There are strong reasons for both options, but I decided to use the SCM data in the current case study. One of the main reasons for this choice was that there are historically many reviews in the partner company done without a tool that captures the necessary data and I wanted to include these reviews in the analysis. This decision has a significant impact on the type of noise occurring in the data. Resulting problems are discussed in Section 15.5.

E.3 Determinining Review Commits

Having decided that all changes done in 'review commits' shall be counted as review remarks, the next question is how to decide whether a commit is a review commit for a certain ticket. The corresponding algorithm is based on the assumption that it is known which commits belong to a certain ticket and that a commit belongs to only one ticket. The partner company has a hook script in the SCM that demands a ticket ID at the start of every commit message, which makes this information easy to extract. Figure E.2 shows how the classification of the commits is done: During its lifetime, a ticket can have one of several states, of which "in implementation" and "in review" are the most important here. "In implementation" means that the code's author is currently working on the ticket, whereas "in review" means that a reviewer is checking the code. Both, changes performed as a reaction to review remarks, as well as changes performed by the

---

[1]CoRT allows reviewers to leave remarks for their on-the-fly fixes (see Chapter 9), but this is used only for a fraction of the fixes.

Table E.1: Comparison of the benefits of the two options to extract review remarks from software repositories

Benefits of extracting the fixed review remarks from the SCM:
The remarks that led to changes in the code base are arguably the most practically relevant.
They can be extracted quite easily and with high accuracy from the SCM, only ticket IDs and time intervals for the reviews are needed.
Gathering complete data is easy.
Remarks that have been communicated only orally will not be missed.
The technique is also usable in settings where review remarks are not stored in a structured form.
Benefits of extracting the review remarks from the review tool database:
The remarks that the reviewer took the time to write down textually are arguably the most practically relevant. (This definition of "practical relevance" would contradict the one given above for SCMs.)
This approach will also cover remarks given only to transport knowledge to the code's author.
The line where a review remark is anchored is probably close to the line that triggered this remark.
One high-level remark can lead to several changes in the code when fixed (like in the example with the typo in the method name given in Section E.1). The review tool stores these conceptual remarks, whereas the SCM stores the low-level changes.
The number of remarks is well-defined, in contrast to the data from the SCM where it is unclear whether two consecutive changed lines should be regarded as one or two remarks (or possibly even more).
Remarks for which the fixing was postponed and not done in the respective ticket can be extracted (also in contrast to the SCM data).
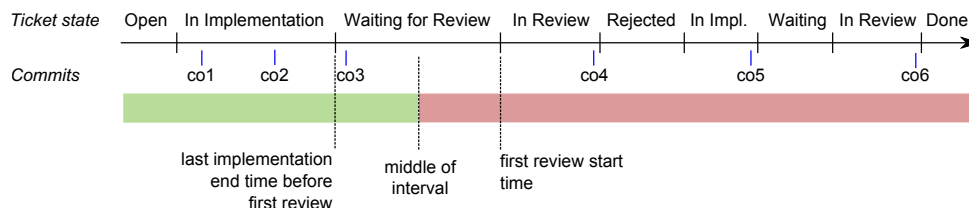
Figure E.2: Example of how to decide which commits for a ticket shall be regarded as "review commits"

reviewer when fixing on-the-fly, shall be included. Therefore, every commit after the start of the first review can be regarded as a 'review commit'. Complications arise because (1) a ticket's state is sometimes changed before the author commits the respective changes (e.g., "co3" in Figure E.2), and because (2) in non-tool reviews, developers sometimes forget to change the ticket state when starting their work and do so later. To deal with these complications, the algorithm heuristically uses the middle of the interval between the end of the last pre-review implementation phase and the start of the first review as the split point.

E.4   Finding Potential Triggers: The RRT Algorithm

Given a set of review commits, the changes (i.e., remarks) in these commits have to be traced back to their potential triggers. This section describes the corresponding algorithm, RRT (for 'review remark tracing'). For now, the finest possible granularity of tracing every changed line is used, as the data can be aggregated later if needed.

As mentioned above, the tracing algorithm is based on the intuition that a remark's trigger

is usually close to the position of the remark. In the simplest case, some line was added or changed during implementation, and some defect in that line was found and fixed in the review phase. This is similar to the SZZ algorithm [353] that is often used in defect prediction studies. But there's a difference to SZZ: In the review case, it is known that a trigger must be one of the changes in the ticket's implementation commits[2]. So in contrast to SZZ, the algorithm cannot stop once it found the previous change of the remark's line (e.g., via 'blame'). If that previous change is not in one of the implementation commits, it needs to skip it and trace back further[3]. The algorithm might find no potential trigger this way, for example, if there was a change in a method during implementation, and in the review it was found that another part of the method has to be changed, too. In this case, the trigger is still close to the remark (i.e., in the same method), but it is not in the same line. This means that the search scope needs to be expanded. These considerations lead to the tracing algorithm outlined in Figure E.3.

As motivated from the example above, the expansion of the search scope should take the structure of the source file into account. I implemented corresponding parsers for Java and XML (and derivatives), the file types most relevant for the case study. If no potential trigger for a change is found in a single line, the algorithm first looks for triggers in the line's block, then in the enclosing blocks, in the containing method, the containing class, until the whole file becomes the search scope. Figure E.4 shows an example of this scope expansion in a Java file. In case there was no implementation commit in the whole file, the algorithm resorts to marking all changes in the whole ticket as potential triggers. For text files not supported by one of the specialized parsers, the algorithm directly moves from the line scope to the file scope.

Another important detail of the algorithm is that it does not stop when the first potential trigger for a remark is found, but instead keeps on searching for further triggers with the same scope. The intuition here is that code might be added in one implementation commit, with a minor change (e.g., fixing a code style issue found by static analysis) done in a later implementation commit. If the algorithm would stop after the first potential trigger, it would only find the minor change, whereas the initial addition of the code is more likely to be the "real" trigger.

To sum up, review remarks can be extracted either from code review repositories or from review commits in the SCM. The thesis focuses on the latter and proposes the RRT algorithm (Figure E.3) to associate review remarks (i.e., changed lines in review commits) with potential triggers (i.e., changed lines in implementation commits for the same ticket). The RRT algorithm finds these triggers by tracing back in history, expanding the search scope if no matching trigger could be found with the smaller scope. It is based on two main assumptions: There exists at least one trigger for each review remark, and the most likely triggers are close[4] to the remark.

E.5 Comparison of RRT to SZZ

The reasoning in Section 15.2 shows that review remark prediction differs from defect prediction. Nevertheless, there are similarities, and it is yet to be shown that the simple SZZ approach [353] that is commonly used for tracing in defect prediction studies is not suitable to determine review remark triggers. I perform SZZ-style tracing (with "git blame") for all review

---

[2]"Trigger" in this case does not necessarily mean that the root cause for the remark (e.g., a defect) was injected with an implementation commit. It just means that the remark was created because of reviewing that part of the code.

[3]Similar to how Kim et al. [199] skip changes that cannot be the cause of a defect in their improved version of SZZ

[4]with closeness defined according to a suitable scope concept for the file type

```
function traceTicket(ticket)
    foreach review commit c in ticket
        traceCommit(c)
    end-foreach
end-function

function traceCommit(commit)
    foreach change part p in commit
        if p is addition of whole file
            assign whole ticket as potential trigger for p
        else if p is deletion of whole file
                or p is rename of whole file without further changes
                or p is change in binary file
                or p shall be forced to file scope (e.g. file too large)
            traceWithScopeAndExpandIfNeeded(create a scope object for the whole file in p)
        else
            traceWithScopeAndExpandIfNeeded(create a line range scope for each single line
                in p)
        end-if
    end-foreach
end-function

function traceWithScopeAndExpandIfNeeded(scope)
    do
        found := traceWithScope(scope)
        if found == AT_LEAST_ONE_TRIGGER_FOUND
            return
        else
            //scope expansion happens depending on the file type
            scope := expand(scope)
    while scope could be expanded

    //the largest (implicit) scope is "whole ticket"
    assign whole ticket as potential trigger for p
end-function

function traceWithScope(scope)
    if scope is line range scope
        prevChange := last change to scope.lineRange before scope.commit
    else
        prevChange := last change to scope.file before scope.commit
    end-if

    if prevChange not found
        return NO_TRIGGER_FOUND
    end-if

    if prevChange is implementation commit for ticket
        assign change parts in prevChange as potential trigger(s) for scope.remark
        traceWithScope(adjust scope to commit before prevChange)
        return AT_LEAST_ONE_TRIGGER_FOUND
    else
        return traceWithScope(adjust scope to commit before prevChange)
    end-if
end-function
```

Figure E.3: RRT algorithm for tracing review remarks to potential triggers (simplified)

```
 1  package a.b.c;
 2
 3  /**
 4   * A class javadoc.
 5   */
 6  public class Xyz {
 7      /**
 8       * One method.
 9       */
10      public void foo() {
11          if (Boolean.getBoolean("prop")) {
12              System.out.println("x");
13              System.out.println("y");
14          }
15          System.out.println("z");
16      }
17
18      /**
19       * Another method.
20       */
21      public void bar() {
22      }
23  }
```

Figure E.4: Example for scope expansion in a Java file, starting with the single line scope in line 12

remarks extracted from the dataset and compare the results to the results of the proposed RRT tracing algorithm.

The analysis is performed on the raw data, i.e., with line granularity. Figure E.5 shows the results. Of 326,066 remark records, only 119,031 (37%) are traced with the same result for the proposed approach and SZZ. For 9,947 (3%), the proposed approach returns further potential triggers because it does not stop at the first candidate. For 197,088 (60%) the results differ completely: For 70,582 (22%) the tracing is stopped prematurely either at a commit from another ticket or at a review commit. For 126,506 (39%) SZZ does not find a trigger because it does not enlarge the search scope when no trigger is found. Figure E.5 also shows that the distribution for the subset of remark lines in Java source files is similar.
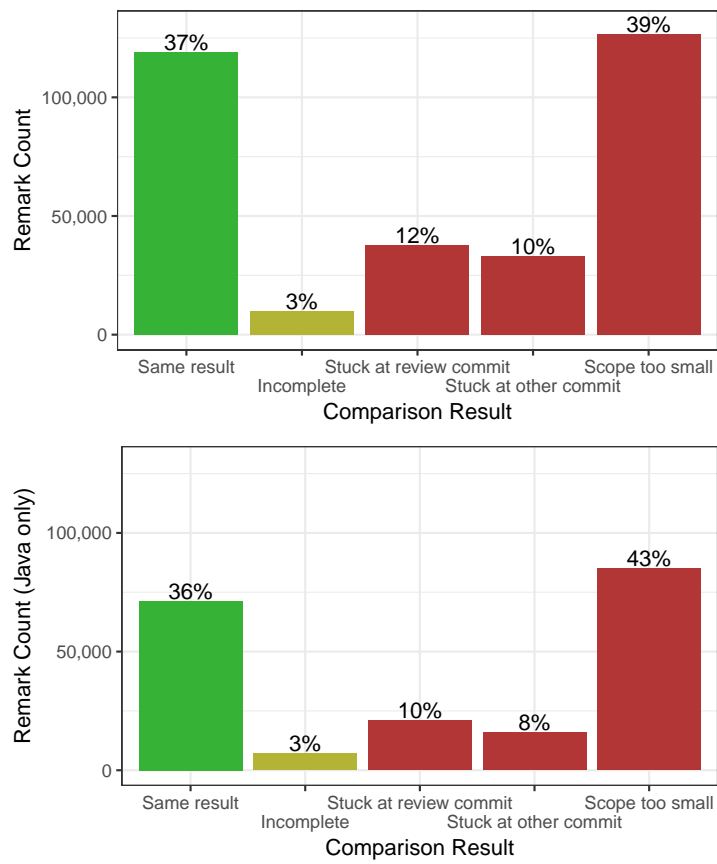
Figure E.5: Comparison of the tracing approach from the current thesis (RRT) and SZZ, both for all remarks (i.e., changed lines/files in review commits) and for remarks in Java files only.

F

Features Used for Classifying Change Parts

Chapter 15's model to predict importance for review uses data from a variety of features. Which features to use was determined systematically, combining results from the literature with an inductive approach.

To identify features from the literature, I looked for defect prediction studies and especially studies that compare the relative suitability of features for defect prediction. I identified these studies using searches on Google Scholar and performed snowballing [405] for further studies, but did not perform a formal systematic literature review. Few studies use change part granularity for prediction, so the features were partly adapted to fit the current context. The studies selected for analysis are: [14, 72, 86, 116, 139, 256, 263, 265, 272, 279, 287, 309, 313, 339, 340, 343, 345, 355].

As second feature source, I used an inductive approach, inspired by qualitative approaches for hypothesis generation [144]: Initially, I sampled some code changes from the data and assigned "codes" to the change parts, similar to "open coding" in qualitative data analysis. Later, I extended the data mining tool with a feature to sample misclassified change parts. These were again analyzed for missing features that could explain the misclassification.

Tables F.1 to F.3 show the final selection of features implemented for the study, with a brief explanation and the source for each feature. 'Source' means the source responsible for the inclusion in this study and does not contain an exhaustive list of all studies that use that feature.

Table F.1: Final selection of change part features used as input for the mining (1/3)

Name and Type	Description	Source / Inspired by
Ticket and commit granularity:		
issueType *(nominal)*	Type of the Jira ticket (e.g., bug or user story task)	[340]
author *(nominal)*	User ID for the author of the commit	inductive
authorDay *(nominal)*	Weekday of the commit	[116, 340]
shiftedAuthorHour *(numeric)*	Hour of the time of the commit. The value is shifted so that 0 stands for 6 AM. In this way, "night" vs "day" can be expressed with a single comparison.	
fileCountInCommit *(numeric)*	Count of files that were changed in the commit.	inductive
hunkCountIn-Commit *(numeric)*	Count of change parts in the commit (changes in binary files count as 1)	inductive
commitContains-Test *(boolean)*	"true" iff the commit contains changes to test code	[313]
File granularity:		
binary *(boolean)*	"true" iff the file is treated as binary. Very large text files ($\geq$ 1 MiB) are also treated as binary.	inductive
filetype *(nominal)*	Extension of the filename (e.g., "java" or "txt")	inductive
srcdir *(nominal)*	Classification of the file in the project: "src" (production code), "test" (test code), "testdata" or "resources"	inductive
project *(nominal)*	Project to which the file belongs. In the partner company, there are also some pseudo projects, e.g., for common build scripts or external dependencies.	inductive
frequentFilename *(nominal)*	Filename (without path, but with extension), but set only when it is one of the 20 most common filenames. In the partner company there are some very common filenames that denote specific roles, e.g., "Messages.java" or "Logger.java".	inductive
fileAgeDays *(numeric)*	Number of days since the creation (initial commit) of the file.	[287, 309, 343]
fileCommitCount *(numeric)*	Number of commits to the file since its creation.	[14, 256, 263, 340]
distinctFileAuthor-Count *(numeric)*	Number of distinct authors of the file since its creation.	[14, 139, 256, 263, 287, 340, 343]
newLineCountIn-File *(numeric)*	Total number of lines in the file (after the commit)	[265, 339]
recentProject-Ownership *(numeric)*	Ratio of the number of commits to the file's project in the last year by the author to the number of commits to the file's project in the last year by all authors.	[272, 355]

Table F.2: Final selection of change part features used as input for the mining (2/3)

Name and Type	Description	Source / Inspired by
File granularity (continued):		
commitsSinceLast-RemarkForAuthor-InProject *(numeric)*	Number of commits since the file's author last received a review remark in the file's project.	[86, 287, 309, 340, 343, 355]
commitsSinceLast-RemarkInFile *(numeric)*	Number of commits since the file last received a review remark.	
hunkCountInFile *(numeric)*	Count of change parts in the file	inductive
changetype *(nominal)*	Git's classification of the change to the file (MODIFY/ADD/RENAME/DELETE/COPY).	inductive
gitSimilarity *(numeric)*	Git's similarity statistic for the file content (i.e., 100 when the content stayed the same)	inductive
newShareOfLines-InFile *(numeric)*	Ratio of the changed lines to the total number of lines (in the new file version).	[313]
isNodeModules *(boolean)*	"true" if the file path denotes a commited external dependency from npm (node.js package manager)	inductive
Change part granularity:		
oldHunkSize *(numeric)*	Number of lines of the change part in the old file version	inductive
newHunkSize *(numeric)*	Number of lines of the change part in the new file version	inductive
changeInHunkSize *(numeric)*	newHunkSize minus oldHunkSize	inductive
commentLine-CountOld *(numeric)*	Number of comment lines in the old side of the change part	[139, 265]
commentLine-CountNew *(numeric)*	Number of comment lines in the new side of the change part	
changeInComment-LineCount *(numeric)*	commentLineCountNew minus commentLineCountOld	
oldBlockCount *(numeric)*	Number of Java blocks (i.e., braces pairs) in the old side of the change part.	[265]
newBlockCount *(numeric)*	Number of Java blocks (i.e., braces pairs) in the new side of the change part.	
changeInBlock-Count *(numeric)*	newBlockCount minus oldBlockCount	

Table F.3: Final selection of change part features used as input for the mining (3/3)

Name and Type	Description	Source / Inspired by
Change part granularity (continued):		
responseForHunk-Old *(numeric)*	RFC metric (Response For a Class; approx. number of distinct method calls) restricted to the code in the old side of the change part	[14, 72, 86, 139, 287]
responseForHunk-New *(numeric)*	RFC metric (Response For a Class; approx. number of distinct method calls) restricted to the code in the new side of the change part	
changeInResponse-ForHunk *(numeric)*	responseForHunkNew minus responseForHunkOld	[14, 86]
whitespaceOnly *(boolean)*	"true" iff only whitespace was changed for the change part.	inductive
packageAndImport-Only *(boolean)*	"true" iff only package declarations and import statements were changed for the change part.	inductive
finalChangeOnly *(boolean)*	"true" iff the change consists only of adding or removing Java's "final" keyword.	inductive
nonnlsChangeOnly *(boolean)*	"true" iff the change consists only of changes to the marker comments and annotations for non-internationalized string constants.	inductive
visibilityChange-Only *(boolean)*	"true" iff only visibility modifiers ("public", "private", . . .) were changed.	inductive
overrideAnnotation *(nominal)*	Denotes which side of the change part contains an "@Override" annotation (none/old/new/both)	inductive
entropyCbMax, entropyCbUp-pQuar, entropyCbMed, entropyCbSum, entropyCbAvg *(numeric)*	This group of features conceptually denotes the "surprisingness" of the new code given the old code base. Formally, the SLP library by Hellendoorn et al. [169] is used to compute an entropy for each token on the new side of the change part, with smaller values for less surprising tokens. These per token entropies are then combined to find the maximum ("Max"), 75% quantile ("UppQuar"), median ("Med"), sum ("UppQuar") and mean ("Avg") for the change part.	inductive, [169]
entropyReMax, entropyReUp-pQuar, entropyReMed, entropyReSum, entropyReAvg *(numeric)*	This group of features conceptually denotes the "surprisingness" of a change part given the earlier change parts under review. Like the "entropyCb. . ." group of features, it uses the SLP library by Hellendoorn et al. [169] and combines the per token values differently for each feature ("Max", "UppQuar", . . .). Pre-processing is performed to make the library work on code changes instead of code; its details can be seen in the study's online material [32].	inductive, [169]

G

Results of the Remark Classification Model for the Training Data

Chapter 15 contains the results of the found rules on the unseen test data. Mainly to identify signs of overfitting, the results on the training data can be useful, too.

Figure G.1 shows projections of the Pareto front obtained by the multi-objective algorithm with and without input from domain experts. It also shows the position of the four selected rules in the objective space. RIPPER and C4.5 are dominated by both Pareto fronts, and do not perform much better, and sometimes worse, than just skipping the review of random change parts. RIPPER differs from all other rulesets by having the form "skip all except . . .". RIPPER and C4.5 were insensitive to the (high) cost of missed remarks and created rulesets with many missed review remarks. The RIPPER and C4.5 rulesets are also more complex than the MO rulesets. The exact numbers for these and the other objectives are shown in Table G.1.

Figure G.1 also allows an estimate of the hardness of the mining task for the objectives. Without missing remarks, the best found model can save the review of 46% of the records. But practically more relevant are the relative numbers of saved Java lines and the trimmed mean of saved records per ticket. Both are much lower (17% and 10%), indicating that the good numbers are due to rare events: tickets with a large number of changes to non-source files. Also, all the stated numbers are optimistic, as they are evaluated on the training set. More realistic results from unseen data follow in Section 15.4.5.

Summing up, the results of the multi-objective rule mining algorithm are better than those obtained with RIPPER and C4.5. The results obtained with user interaction are better in the regions of the objective space that were focused on by the user, i.e., complexity and broad ticket coverage. Figure 15.6 shows the ruleset that came out of this interaction between mining tool and domain experts. 84% of the savings of this ruleset are due to three simple syntactic rules.
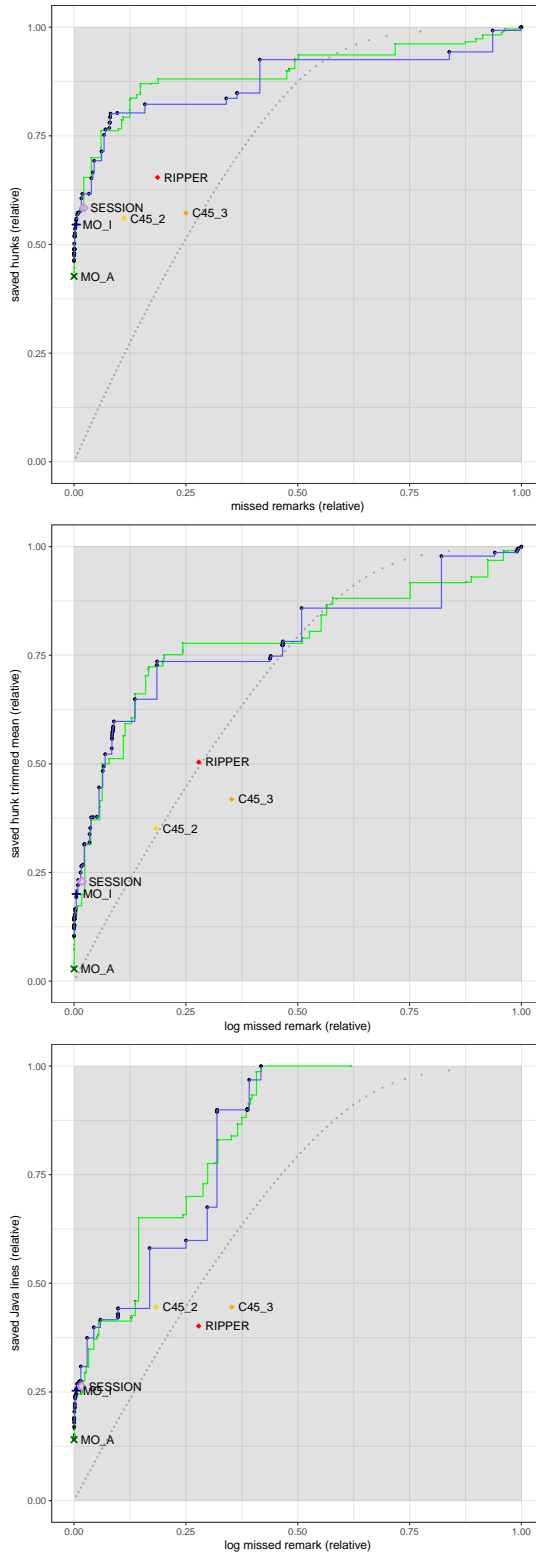
Figure G.1: Pareto fronts and selected rulesets, evaluated on the training data. The plots show two-dimensional projections from the seven-dimensional objective space. The gray dots show the baseline performance of leaving out a certain percentage of records per ticket; each dot corresponds to a percentage value, with results averaged over 100 random seeds.

Table G.1: Objective values for the selected rulesets on the training data.

Ruleset	Objectives to Minimize				Objectives to Maximize		
	Compl-exity	Feature Count	Missed Remark Count	Log-Transf. Missed Remarks	Saved Record Count	Tr.M.[1] Saved Records Per Ticket	Saved LOC in Java Files
SESSION	40	17	1,500	142.4	410,974	10.1	303,943
MO_I	58	17	338	40.6	384,058	8.8	292,371
MO_A	184	24	0	0.0	300,266	1.2	161,920
RIPPER_S[2]	200	17	16,806	2,493.6	445,275	21.6	431,981
RIPPER	342	25	12,867	2,245.3	460,408	22.1	464,148
C4.5_2	151,084	39	7,669	1,476.6	394,179	15.4	515,225
C4.5_3	63,872	38	17,231	2,838.4	402,585	18.4	514,911
Max. Value[3]	∞	52	68,960	8,055.8	703,706	43.9	1,155,636

[1] Tr.M. := trimmed mean

[2] During the team session it was decided to remove one further ticket from the training data. RIPPER is the rule set learned with the final data, RIPPER_S is based on the older data and was used in the survey.

[3] The last row shows the total count / maximum possible value for the respective objective.

Bibliography

[1] Z. Abdelnabi, G. Cantone, M. Ciolkowski, and D. Rombach. Comparing code reading techniques applied to object-oriented software frameworks with regard to effectiveness and defect detection rate. In *Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on*, pages 239–248. IEEE, 2004.

[2] S. Adolph, W. Hall, and P. Kruchten. Using grounded theory to study the experience of software development. *Empirical Software Engineering*, 16(4):487–513, 2011.

[3] A. Agresti. *An introduction to categorical data analysis*. Wiley, 2nd edition, 2007.

[4] A. Agresti. *Analysis of Ordinal Categorical Data*. Wiley, 2nd edition, 2010.

[5] Ö. Albayrak and D. Davenport. Impact of maintainability defects on code inspections. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, page 50. ACM, 2010.

[6] C. Alexander. *The timeless way of building*. Oxford University Press, 1979.

[7] N. B. Ali and K. Petersen. A consolidated process for software process simulation: state of the art and industry experience. In *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*, pages 327–336. IEEE, 2012.

[8] N. B. Ali, K. Petersen, and C. Wohlin. A systematic literature review on the industrial use of software process simulation. *Journal of Systems and Software*, 97:65–85, 2014. DOI: `10.1016/j.jss.2014.06.059`.

[9] D. Anderson. *Kanban: Successful Evolutionary Change for Your Technology Business*. Blue Hole Press, 2010. ISBN: 9780984521401.

[10] D. J. Anderson, G. Concas, M. I. Lunesu, M. Marchesi, and H. Zhang. A comparative study of scrum and kanban approaches on a real case study using simulation. In *Agile Processes in Software Engineering and Extreme Programming*, pages 123–137. Springer, 2012.

[11] P. Anderson, T. Reps, and T. Teitelbaum. Design and implementation of a fine-grained software inspection tool. *Software Engineering, IEEE Transactions on*, 29(8):721–733, 2003.

[12] V. Anu, G. Walia, W. Hu, J. C. Carver, and G. Bradshaw. Using a cognitive psychology perspective on errors to improve requirements quality: an empirical investigation. In *Software Reliability Engineering (ISSRE), 2016 IEEE 27th International Symposium on*, pages 65–76. IEEE, 2016.

[13] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 2–13. IEEE Computer Society, 2004. DOI: 10.1109/ASE.2004.1342719.

[14] E. Arisholm, L. C. Briand, and E. B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83(1):2–17, 2010.

[15] J. Asundi and R. Jayant. Patch review processes in open source software development communities: a comparative case study. In *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, pages 166c–166c. IEEE, 2007.

[16] B. Athreya and C. Scaffidi. Towards aiding within-patch information foraging by end-user programmers. In *Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on*, pages 13–20. IEEE, 2014.

[17] A. Aurum, H. Petersson, and C. Wohlin. State-of-the-art: software inspections after 25 years. *Software Testing, Verification and Reliability*, 12(3):133–154, 2002.

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

[19] R. A. Baker Jr. Code reviews enhance software quality. In *Proceedings of the 19th International Conference on Software Engineering*, pages 570–571. ACM, 1997.

[20] V. Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 931–940. IEEE Press, 2013.

[21] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri. Helping developers help themselves: automatic decomposition of code review changesets. In *Proceedings of the 2015 International Conference on Software Engineering. IEEE Press*, 2015.

[22] K. Barton. *MuMIn: Multi-Model Inference*. R package version 1.42.1. 2018. URL: https://CRAN.R-project.org/package=MuMIn.

[23] V. R. Basili. Evolving and packaging reading technologies. *Journal of Systems and Software*, 38(1):3–12, 1997.

[24] V. R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sørumgård, and M. V. Zelkowitz. The empirical investigation of perspective-based reading. *Empirical Software Engineering*, 1(2):133–164, 1996.

[25] V. Basili, G. Caldiera, F. Lanubile, and F. Shull. Studies on reading techniques. In *Proc. of the Twenty-First Annual Software Engineering Workshop*, volume 96, page 002, 1996.

[26] D. Bates, M. Maechler, B. Bolker, S. Walker, et al. Lme4: linear mixed-effects models using eigen and s4. *R package version*, 1(7):1–23, 2014.

[27] T. Baum. Detailed table with review effects (team level) and their connections to contextual factors and process variants for "factors influencing code review processes in industry". 2016. URL: http://dx.doi.org/10.6084/m9.figshare.5104111.

[28] T. Baum. Leveraging pre-commit hooks for context-sensitive checklists: a case study. In *Fachtagung des GI-Fachbereichs Softwaretechnik, Software Engineering (SE 2015), Dresden, Germany*, pages 219–222, 2015.

[29]   T. Baum, A. Bacchelli, and K. Schneider. Associating working memory capacity and code change ordering with code review performance. *Empirical Software Engineering*, 2018. DOI: `10.1007/s10664-018-9676-8`.

[30]   T. Baum, S. Herbold, and K. Schneider. A multi-objective anytime rule mining system to ease iterative feedback from domain experts. *arXiv preprint arXiv:1812.09746*, 2018.

[31]   T. Baum, S. Herbold, and K. Schneider. An industrial case study on shrinking code review changesets through remark prediction. *arXiv preprint arXiv:1812.09510*, 2018.

[32]   T. Baum, S. Herbold, and K. Schneider. Online appendix for "an industrial case study on shrinking code review changesets through remark prediction". 2018. URL: `http://dx.doi.org/10.6084/m9.figshare.7438676`.

[33]   T. Baum and F. Kortum. Simulation results and source code for pre/post commit comparison paper. `https://github.com/FG-SE/PrePostReviewProcessSimulation/releases/tag/v1.0`.

[34]   T. Baum, F. Kortum, K. Schneider, A. Brack, and J. Schauder. Comparing pre commit reviews and post commit reviews using process simulation. In *Software and System Process (ICSSP), 2016 International Conference on*, Austin, TX, USA, 2016. DOI: `10.1109/ICSSP.2016.012`.

[35]   T. Baum, F. Kortum, K. Schneider, A. Brack, and J. Schauder. Comparing pre-commit reviews and post-commit reviews using process simulation. *Journal of Software: Evolution and Process*, 29(11):e1865, 2017. DOI: `10.1002/smr.1865`.

[36]   T. Baum, H. Leßmann, and K. Schneider. Online material for survey on code review use. DOI: `10.6084/m9.figshare.5104249`. URL: `http://dx.doi.org/10.6084/m9.figshare.5104249`.

[37]   T. Baum, H. Leßmann, and K. Schneider. The choice of code review process: a survey on the state of the practice. In M. Felderer, D. Méndez Fernández, B. Turhan, M. Kalinowski, F. Sarro, and D. Winkler, editors, *Product-Focused Software Process Improvement*, pages 111–127, Cham. Springer International Publishing, 2017. ISBN: 978-3-319-69926-4. DOI: `10.1007/978-3-319-69926-4\_9`.

[38]   T. Baum, O. Liskin, K. Niklas, and K. Schneider. A faceted classification scheme for change-based industrial code review processes. In *Software Quality, Reliability and Security (QRS), 2016 IEEE International Conference on*, pages 74–85, Vienna, Austria. IEEE, 2016. DOI: `10.1109/QRS.2016.19`.

[39]   T. Baum, O. Liskin, K. Niklas, and K. Schneider. Factors influencing code review processes in industry. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 85–96, Seattle, WA, USA. ACM, 2016. ISBN: 978-1-4503-4218-6. DOI: `10.1145/2950290.2950323`.

[40]   T. Baum and K. Schneider. On the need for a new generation of code review tools. In *Product-Focused Software Process Improvement: 17th International Conference, PROFES 2016, Trondheim, Norway, November 22-24, 2016, Proceedings 17*, pages 301–308. Springer, 2016. DOI: `10.1007/978-3-319-49094-6\_19`.

[41]   T. Baum, K. Schneider, and A. Bacchelli. On the optimal order of reading source code changes for review. In *33rd IEEE International Conference on Software Maintenance and Evolution (ICSME), Proceedings*, pages 329–340, 2017. DOI: `10.1109/ICSME.2017.28`.

[42] T. Baum, K. Schneider, and A. Bacchelli. Online material for "Associating working memory capacity and code change ordering with code review performance". 2018. URL: http://dx.doi.org/10.6084/m9.figshare.5808609.

[43] T. Baum, K. Schneider, and A. Bacchelli. Online material for "On the optimal order of reading source code changes for review". 2017. URL: http://dx.doi.org/10.6084/m9.figshare.5236150.

[44] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey. Investigating technical and non-technical factors influencing modern code review. *Empirical Software Engineering*, 21:932–959, 2016. DOI: 10.1007/s10664-015-9366-8.

[45] A. Begel and H. Vrzakova. Eye movements in code review. In *Proceedings of the Workshop on Eye Movements in Programming*, page 5. ACM, 2018. DOI: 10.1145/3216723.3216727.

[46] F. Belli and R. Crisan. Empirical performance analysis of computer-supported code-reviews. In *Software Reliability Engineering, 1997. Proceedings., The Eighth International Symposium on*, pages 245–255. IEEE, 1997.

[47] G. R. Bergersen and J.-E. Gustafsson. Programming skill, knowledge, and working memory among professional software developers from an investment theory perspective. *Journal of Individual Differences*, 32(4):201–209, 2011.

[48] M. Bernhart, A. Mauczka, and T. Grechenig. Adopting code reviews for agile software development. In *Agile Conference (AGILE), 2010*, pages 44–47. IEEE, 2010. DOI: 10.1109/AGILE.2010.18.

[49] M. Bernhart, S. Reiterer, K. Matt, A. Mauczka, and T. Grechenig. A task-based code review process and tool to comply with the do-278/ed-109 standard for air traffic managment software development: an industrial case study. In *High-Assurance Systems Engineering (HASE), 2011 IEEE 13th International Symposium on*, 2011. DOI: 10.1109/HASE.2011.54.

[50] M. Bernhart, S. Strobl, A. Mauczka, and T. Grechenig. Applying continuous code reviews in airport operations software. In *Quality Software (QSIC), 2012 12th International Conference on*, pages 214–219. IEEE, 2012. DOI: 10.1109/QSIC.2012.61.

[51] B. Biegel, F. Beck, W. Hornig, and S. Diehl. The order of things: how developers sort fields and methods. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 88–97. IEEE, 2012.

[52] S. Biffl. Analysis of the impact of reading technique and inspector capability on individual inspection performance. In *Software Engineering Conference, 2000. APSEC 2000. Proceedings. Seventh Asia-Pacific*, pages 136–145. IEEE, 2000.

[53] S. Biffl, P. Grünbacher, and M. Halling. A family of experiments to investigate the effects of groupware for software inspection. *Automated Software Engineering*, 13(3):373–394, 2006. DOI: 10.1007/s10851-006-8531-5.

[54] S. Biffl and M. Halling. Investigating the influence of inspector capability factors with four inspection techniques on inspection performance. In *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on*, pages 107–117. IEEE, 2002.

[55] C. Bird, T. Carnahan, and M. Greiler. Lessons learned from building and deploying a code review analytics platform. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 191–201, 2015.

[56] D. B. Bisant and J. R. Lyle. A two-person inspection method to improve programming productivity. *IEEE Transactions on Software Engineering*, 15(10):1294–1304, 1989.

[57] J. B. Black and G. H. Bower. Story understanding as problem-solving. *Poetics*, 9(1-3):223–250, 1980.

[58] A. Bosu and J. C. Carver. Impact of peer code review on peer impression formation: a survey. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*, pages 133–142. IEEE, 2013.

[59] A. Bosu and J. C. Carver. Peer code review in open source communitiesusing reviewboard. In *Proceedings of the ACM 4th annual workshop on Evaluation and usability of programming languages and tools*, pages 17–24. ACM, 2012.

[60] A. Bosu, J. C. Carver, C. Bird, J. Orbeck, and C. Chockley. Process aspects and social dynamics of contemporary code review: insights from open source development as well as industrial practice at microsoft. *IEEE Transactions on Software Engineering*, 2016. DOI: 10.1109/TSE.2016.2576451.

[61] A. Bosu, M. Greiler, and C. Bird. Characteristics of useful code reviews: an empirical study at microsoft. In *MSR '15 Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 146–156, 2015.

[62] L. R. Brothers. Multimedia groupware for code inspection. In *Communications, 1992. ICC'92, Conference record, SUPERCOMM/ICC'92, Discovering a New World of Communications., IEEE International Conference on*, pages 1076–1081. IEEE, 1992.

[63] L. Brothers, V. Sembugamoorthy, and M. Muller. Icicle: groupware for code inspection. In *Proceedings of the 1990 ACM conference on Computer-supported cooperative work*, pages 169–181. ACM, 1990. DOI: 10.1145/99332.99353.

[64] L. Brothers, V. Sembugamoorthy, and A. E. Irgon. Knowledge-based code inspection with icicle. In *IAAI*, pages 295–314, 1992.

[65] B. Brykczynski. A survey of software inspection checklists. *ACM SIGSOFT Software Engineering Notes*, 24(1):82, 1999.

[66] R. P. Buse and W. R. Weimer. Automatically documenting program changes. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 33–42. ACM, 2010.

[67] D. Caivano, F. Lanubile, and G. Visaggio. Scaling up distributed software inspections. In *Proceedings of the Fourth ICSE Workshop on Software Engineering over the Internet*, 2001.

[68] G. Canfora, A. D. Lucia, M. D. Penta, R. Oliveto, A. Panichella, and S. Panichella. Defect prediction as a multiobjective optimization problem. *Software Testing, Verification and Reliability*, 25(4):426–459, 2015.

[69] K. Chan. An agent-based approach to computer assisted code inspections. In *Software Engineering Conference, 2001. Proceedings. 2001 Australian*, pages 147–152. IEEE, 2001.

[70] L. Chan, K. Jiang, and S. Karunasekera. A tool to support perspective based approach to software code inspection. In *Software Engineering Conference, 2005. Proceedings. 2005 Australian*, pages 110–117. IEEE, 2005. DOI: `10.1109/ASWEC.2005.10`.

[71] K. Charmaz. *Constructing Grounded Theory*. Introducing Qualitative Methods series. SAGE Publications, 2014. ISBN: 9781446293492.

[72] X. Chen, Y. Shen, Z. Cui, and X. Ju. Applying feature selection to software defect prediction using multi-objective optimization. In *Computer Software and Applications Conference (COMPSAC), 2017 IEEE 41st Annual*, volume 2, pages 54–59. IEEE, 2017.

[73] X. Chen, Y. Zhao, Q. Wang, and Z. Yuan. Multi: multi-objective effort-aware just-in-time software defect prediction. *Information and Software Technology*, 93:1–13, 2018.

[74] H. ÇİFCİ. *A Workflow Based Online Software Review System*. Master's thesis, Middle East Technical University, 2004.

[75] M. Ciolkowski, O. Laitenberger, and S. Biffl. Software reviews: the state of the practice. *IEEE Software*, 20(6):46–51, 2003.

[76] P. Clarke and R. V. O'Connor. The situational factors that affect the software development process: towards a comprehensive reference framework. *Information and Software Technology*, 54(5):433–447, 2012.

[77] J. Cohen. *Statistical power analysis for the behavioral sciences. Revised edition*. Academic Press, 1977.

[78] W. W. Cohen. Fast effective rule induction. In *Twelfth International Conference on Machine Learning*, pages 115–123. Morgan Kaufmann, 1995.

[79] G. Coleman and R. O'Connor. Using grounded theory to understand software process improvement: a study of irish software product companies. *Information and Software Technology*, 49(6):654–667, 2007.

[80] G. Coleman and R. V. O'Connor. An investigation into software development process formation in software start-ups. *Journal of Enterprise Information Management*, 21(6):633–648, 2008.

[81] J. Corbin and A. Strauss. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE Publications, 3e edition, 2007. ISBN: 9781452278933.

[82] N. Cowan. The magical mystery four: how is working memory capacity limited, and why? *Current directions in psychological science*, 19(1):51–57, 2010.

[83] I. Crk, T. Kluthe, and A. Stefik. Understanding programming expertise: an empirical study of phasic brain wave changes. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 23(1):2, 2016.

[84] M. Csikszentmihalyi. *Flow: The Psychology of Optimal Experience*. Harper Perennial Modern Classics, 2008.

[85] J. Czerwonka, M. Greiler, and J. Tilford. Code reviews do not find bugs. how the current code review best practice slows us down. In *Proceedings of the 2015 International Conference on Software Engineering – Volume 2*. IEEE, May 2015. URL: `http://research.microsoft.com/apps/pubs/default.aspx?id=242201`.

[86] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 31–41. IEEE, 2010.

[87] M. D'Ambros, M. Lanza, and R. Robbes. Commit 2.0. In *Proceedings of the 1st Workshop on Web 2.0 for Software Engineering*, pages 14–19. ACM, 2010.

[88] H. K. Dam, T. Tran, and A. Ghose. Explainable software analytics. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 53–56. ACM, 2018.

[89] M. Daneman and P. A. Carpenter. Individual differences in working memory and reading. *Journal of verbal learning and verbal behavior*, 19(4):450–466, 1980.

[90] M. Daneman and P. M. Merikle. Working memory and language comprehension: a meta-analysis. *Psychonomic bulletin & review*, 3(4):422–433, 1996.

[91] J. P. Davis, K. M. Eisenhardt, and C. B. Bingham. Developing theory through simulation methods. *Academy of Management Review*, 32(2):480–499, 2007.

[92] A. B. De Carvalho, A. Pozo, and S. R. Vergilio. A symbolic fault-prediction model based on multiobjective particle swarm optimization. *Journal of Systems and Software*, 83(5):868–882, 2010.

[93] A. De Lucia, F. Fasano, G. Scanniello, and G. Tortora. Evaluating distributed inspection through controlled experiments. *IET software*, 3(5):381–394, 2009. DOI: `10.1049/iet-sen.2008.0101`.

[94] A. De Lucia, F. Fasano, G. Scanniello, and G. Tortora. Improving artefact quality management in advanced artefact management system with distributed inspection. *IET software*, 5(6):510–527, 2011. DOI: `10.1049/iet-sen.2010.0108`.

[95] A. De Lucia, F. Fasano, G. Tortora, and G. Scanniello. Assessing the effectiveness of a distributed method for code inspection: a controlled experiment. In *Global Software Engineering, 2007. ICGSE 2007. Second IEEE International Conference on*, pages 252–261. IEEE, 2007. DOI: `10.1109/ICGSE.2007.11`.

[96] A. De Lucia, F. Fasano, G. Tortora, and G. Scanniello. Integrating a distributed inspection tool within an artefact management system. In *ICSOFT 2007 – 2nd International Conference on Software and Data Technologies, Proceedings*, pages 184–189, Dec. 2007.

[97] B. B. N. de França and G. H. Travassos. Simulation based studies in software engineering: a matter of validity. *CLEI Electronic Journal*, 18(1):5–5, 2015.

[98] G.-J. de Vreede, P. G. Koneri, D. L. Dean, A. L. Fruhling, and P. Wolcott. A collaborative software code inspection: the design and evaluation of a repeatable collaboration process in the field. *International Journal of Cooperative Information Systems*:205–228, 2006. DOI: `10.1142/S0218843006001347`.

[99] C. Denger, M. Ciolkowski, and F. Lanubile. Does active guidance improve software inspections? a preliminary empirical study. In *IASTED Conf. on Software Engineering*, pages 408–413, 2004.

[100] C. Denger and F. Shull. A practical approach for quality-driven inspections. *Software, IEEE*, 24(2):79–86, 2007.

[101] D. DeStefano and J.-A. LeFevre. Cognitive load in hypertext reading: a review. *Computers in human behavior*, 23(3):1616–1641, 2007.

[102] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse. Untangling fine-grained code changes. In *Software Analysis, Evolution and Reengineering, 2015 IEEE 22nd International Conference on*, pages 341–350. IEEE, 2015.

[103] B. Doherty and S. Sahibuddin. Software quality through distributed code inspection. *Proceedings of the International Conference on Software Quality Engineering, SQE*:159–168, Jan. 1997. DOI: `10.2495/SQE970151`.

[104] E. W. dos Santos and I. Nunes. Investigating the effectiveness of peer code review in distributed software development. In *Proceedings of the 31st Brazilian Symposium on Software Engineering*, pages 84–93. ACM, 2017. DOI: `10.1145/3131151.3131161`.

[105] J. Dowell and J. Long. Target paper: conception of the cognitive engineering design problem. *Ergonomics*, 41(2):126–139, 1998.

[106] J. Drake, V. Mashayekhi, J. Riedl, and W.-T. Tsai. A distributed collaborative software inspection tool: design, prototype, and early trial. In *Proceedings of the 30th Aerospace Sciences Conference*, 1991.

[107] A. Dunsmore, M. Roper, and M. Wood. Object-oriented inspection in the face of delocalisation. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 467–476. ACM, 2000.

[108] A. Dunsmore, M. Roper, and M. Wood. Practical code inspection for object-oriented systems. In *WISE*, volume 1, pages 49–57, 2001.

[109] A. Dunsmore, M. Roper, and M. Wood. Systematic object-oriented inspection – an empirical study. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 135–144. IEEE Computer Society, 2001.

[110] A. Dunsmore, M. Roper, and M. Wood. The development and evaluation of three diverse techniques for object-oriented code inspection. *Software Engineering, IEEE Transactions on*, 29(8):677–686, 2003. DOI: `10.1109/TSE.2003.1223643`.

[111] A. Dunsmore, M. Roper, and M. Wood. The role of comprehension in software inspection. *Journal of Systems and Software*, 52(2):121–129, 2000.

[112] T. Dürschmid. Continuous code reviews. In *Companion to the first International Conference on the Art, Science and Engineering of Programming*, 2017. DOI: `10.1145/3079368.3079374`.

[113] N. Dzamashvili-Fogelström and T. Gorschek. Test-case driven versus checklist-based inspections of software requirements–an experimental evaluation. In *10th Workshop on Requirements Engineering (WER 07)*, 2007.

[114] F. Elberzhager, R. Eschbach, and J. Kloos. Indicator-based inspections: a risk-oriented quality assurance approach for dependable systems. In *Fachtagung des GI-Fachbereichs Softwaretechnik, Software Engineering (SE 2010), Paderborn, Germany*, 2010.

[115] K. A. Ericsson, R. T. Krampe, and C. Tesch-Römer. The role of deliberate practice in the acquisition of expert performance. *Psychological review*, 100(3):363, 1993.

[116] J. Eyolfson, L. Tan, and P. Lam. Do time of day and developer experience affect commit bugginess? In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 153–162. ACM, 2011.

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

[118] D. Falessi, N. Juristo, C. Wohlin, B. Turhan, J. Münch, A. Jedlitschka, and M. Oivo. Empirical software engineering experts on the use of students and professionals in experiments. *Empirical Software Engineering*:1–38, 2017.

[119] E. Farchi and B. R. Harrington. Assisting the code review process using simple pattern recognition. In *Hardware and Software, Verification and Testing*, pages 103–115. Springer, 2006.

[120] J.-M. Favre. Understanding-in-the-large. In *Program Comprehension, 1997. IWPC'97. Proceedings., Fifth Iternational Workshop on*, pages 29–38. IEEE, 1997.

[121] M. Fejzer, M. Wojtyna, M. Burzańska, P. Wiśniewski, and K. Stencel. Supporting code review by automatic detection of potentially buggy changes. In *International Conference: Beyond Databases, Architectures and Structures*, pages 473–482. Springer, 2015.

[122] A. L. Ferreira, R. J. Machado, L. Costa, J. G. Silva, R. F. Batista, and M. C. Paulk. An approach to improving software inspections performance. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–8. IEEE, 2010.

[123] A. Field and G. Hole. *How to design and report experiments*. Sage, 2002.

[124] B. Floyd, T. Santander, and W. Weimer. Decoding the representation of code in the brain: an fmri study of code review and expertise. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*, 2017. DOI: `10.1109/ICSE.2017.24`.

[125] B. Fluri, M. Wursch, M. PInzger, and H. C. Gall. Change distilling: tree differencing for fine-grained source code change extraction. *Software Engineering, IEEE Transactions on*, 33(11):725–743, 2007.

[126] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*:1–49, 2015.

[127] Forrester Research, Inc. The value and importance of code reviews. Mar. 2010. URL: `http://embedded-computing.com/white-papers/white-paper-value-importance-code-reviews/`. last checked 2017-06-13.

[128] L. T. Frase. Paragraph organization of written materials: the influence of conceptual clustering upon the level and organization of recall. *Learning and instructional Processes*, 1969.

[129] E. Fregnan. *Automatic Ordering of Code Changes for Review*. Master's thesis, TU Delft, 2018.

[130] E. Fregnan, T. Baum, F. Palomba, and A. Bacchelli. A survey on software coupling relations and tools. *Information and Software Technology*, 107:159–178, 2019. DOI: `https://doi.org/10.1016/j.infsof.2018.11.008`.

[131] S. Friese. *Qualitative Data Analysis with ATLAS.ti*. SAGE Publications, 2012. ISBN: 9780857021311.

[132] T. Fritz, D. C. Shepherd, K. Kevic, W. Snipes, and C. Bräunlich. Developers' code context models for change tasks. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 7–18. ACM, 2014.

[133] W. Fu, T. Menzies, D. Chen, and A. Agrawal. Building better quality predictors using "ϵ-dominance". *arXiv preprint arXiv:1803.04608*, 2018.

[134] L. Gasparini. *Visualisation of Code Changes for Code Review*. Master's thesis, Delft University of Technology, 2019.

[135] X. Ge. *Improving Tool Support for Software Developers through Refactoring Detection*. PhD thesis, North Carolina State University, 2014.

[136] X. Ge, S. Sarkar, J. Witschey, and E. Murphy-Hill. Refactoring-aware code review. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2017. DOI: `10.1109/VLHCC.2017.8103453`.

[137] Y. Geffen and S. Maoz. On method ordering. In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*, pages 1–10, 2016. DOI: `10.1109/ICPC.2016.7503711`.

[138] Ç. E. Gerede and Z. Mazan. Will it pass? predicting the outcome of a source code review. *Turkish Journal of Electrical Engineering & Computer Sciences*, 26(3):1343–1353, 2018.

[139] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall. Method-level bug prediction. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 171–180. ACM, 2012.

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

[141] J. Gintell, J. Arnold, M. Houde, J. Kruszelnicki, R. McKenney, and G. Memmi. Scrutiny: a collaborative inspection and review system. In *European Software Engineering Conference*, pages 344–360. Springer, 1993. DOI: `10.1007/3-540-57209-0_24`.

[142] J. Gintell, M. Houde, and R. McKenney. Lessons learned by building and using scrutiny, a collaborative software inspection system. In *Proceedings of the International Workshop on Computer-Aided Software Engineering*, pages 350–357. IEEE, Jan. 1995. DOI: `10.1109/CASE.1995.465299`.

[143] B. G. Glaser. *Theoretical Sensitivity – Advances in the Methodology of Grounded Theory*. The Sociology Press, 1978.

[144] B. G. Glaser and A. L. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine, 1967.

[145] V. U. Gómez, S. Ducasse, and T. D'Hondt. Visually characterizing source code changes. *Science of Computer Programming*, 98:376–393, 2015.

[146] S. Göpferich. Comprehensibility assessment using the Karlsruhe comprehensibility concept. *The Journal of Specialised Translation*, 11(2009):31–52, 2009.

[147] G. Gousios, M. Pinzger, and A. v. Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, pages 345–355, Hyderabad, India. ACM, 2014.

[148] A. C. Graesser, D. S. McNamara, and M. M. Louwerse. What do readers need to learn in order to process coherence relations in narrative and expository text. *Rethinking reading comprehension*:82–98, 2003.

[149] P. Green II, T. Menzies, S. Williams, and O. El-Rawas. Understanding the value of software engineering technologies. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 52–61. IEEE Computer Society, 2009.

[150] R. Gripp. *Automatische Erzeugung von Zusammenfassungen für Quellcode-Änderungen*. Master's thesis, Leibniz Universität Hannover, 2018.

[151] P. Grünbacher, M. Halling, and S. Biffl. An empirical study on groupware support for software inspection meetings. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 4–11. IEEE, 2003. DOI: 10.1109/ASE.2003.1240289.

[152] C. Guinan and A. F. Smeaton. Information retrieval from hypertext using dynamically planned guided tours. In *Proceedings of the ACM conference on Hypertext*, pages 122–130. ACM, 1992.

[153] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *Software Engineering, IEEE Transactions on*, 38(6):1276–1304, 2012. DOI: 10.1109/TSE.2011.103.

[154] M. Halling, S. Biffl, and P. Grünbacher. A groupware-supported inspection process for active inspection management. In *Euromicro Conference, 2002. Proceedings. 28th*, pages 251–258. IEEE, 2002. DOI: 10.1109/EURMIC.2002.1046168.

[155] M. Halling, S. Biffl, and P. Grünbacher. An experiment family to investigate the defect detection effect of tool-support for requirements inspection. In *Software Metrics Symposium, 2003. Proceedings. Ninth International*, pages 278–285. IEEE, 2003. DOI: 10.1109/METRIC.2003.1232474.

[156] M. Halling, P. Grünbacher, and S. Biffl. Tailoring a cots group support system for software requirements inspection. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 201–208. IEEE, 2001. DOI: 10.1109/ASE.2001.989806.

[157] N. Hammond and L. Allinson. Travel around a learning support environment: rambling, orienteering or touring? In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 269–273. ACM, 1988.

[158] B. Hanington and B. Martin. *Universal methods of design: 100 ways to research complex problems, develop innovative ideas, and design effective solutions*. Rockport Publishers, 2012.

[159] L. Harjumaa, H. Hedberg, and I. Tervonen. A path to virtual software inspection. In *Quality Software, 2001. Proceedings. Second Asia-Pacific Conference on*, pages 283–287. IEEE, 2001. DOI: 10.1109/APAQS.2001.990032.

[160] L. Harjumaa and I. Tervonen. Virtual software inspections over the internet. In *Proc. of the Third ICSE Workshop on Software Engineering over the Internet*, 2000.

[161] L. Harjumaa and I. Tervonen. Www-based tool for software inspection. In *Proceedings of the Hawaii International Conference on System Sciences*, pages 379–388. Institute of Electrical and Electronics Engineers Computer SocietyLos Alamitos, CA, United States, Jan. 1998. DOI: 10.1109/HICSS.1998.656308.

[162] L. Harjumaa, I. Tervonen, and A. Huttunen. Peer reviews in real life-motivators and demotivators. In *Quality Software, 2005.(QSIC 2005). Fifth International Conference on*, pages 29–36. IEEE, 2005.

[163] L. Harjumaa, I. Tervonen, and P. Vuorio. Improving software inspection process with patterns. In *Quality Software, 2004. QSIC 2004. Proceedings. Fourth International Conference on*, pages 118–125. IEEE, 2004. DOI: `10.1109/QSIC.2004.1357952`.

[164] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88. IEEE Computer Society, 2009.

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

[166] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram. Advancing candidate link generation for requirements tracing: the study of methods. *IEEE Transactions on Software Engineering*, 32(1):4–19, 2006.

[167] H. Hedberg. Introducing the next generation of software inspection tools. In *Product Focused Software Process Improvement*, pages 234–247. Springer, 2004. DOI: `10.1007/978-3-540-24659-6_17`.

[168] H. Hedberg and L. Harjumaa. Virtual software inspections for distributed software engineering projects. In *Proceedings of ICSE International Workshop on Global Software Development*. Citeseer, 2002.

[169] V. J. Hellendoorn and P. Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 763–773. ACM, 2017.

[170] A. Z. Henley, K. Muslu, M. Christakis, S. D. Fleming, and C. Bird. Cfar: a tool to increase communication, productivity, and review quality in collaborative code review. In *CHI '18 Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 2018. DOI: `10.1145/3173574.3173731`.

[171] M. Hentschel, R. Hähnle, and R. Bubel. Can formal methods improve the efficiency of code reviews? In *International Conference on Integrated Formal Methods*, pages 3–19. Springer, 2016.

[172] K. Herzig and A. Zeller. The impact of tangled code changes. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 121–130. IEEE, 2013.

[173] A. R. Hevner. A three cycle view of design science research. *Scandinavian journal of information systems*, 19(2):4, 2007.

[174] G. J. Holzmann. Scrub: a tool for code reviews. *Innovations in Systems and Software Engineering*, 6(4):311–318, 2010. DOI: `10.1007/s11334-010-0136-x`.

[175] Y. Huang, N. Jia, X. Chen, K. Hong, and Z. Zheng. Salient-class location: help developers understand code change in code review. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 770–774. ACM, 2018.

[176] J. Humble and D. Farley. *Continuous delivery*. Addison-Wesley, 2011.

[177] B. C. Hungerford, A. R. Hevner, and R. W. Collins. Reviewing software diagrams: a cognitive study. *IEEE Transactions on Software Engineering*, 30(2):82–96, 2004.

[178] Systems and software engineering Vocabulary ISO/IEC/IEEE 24765: 2010. Standard 24765, ISO/IEC/IEEE, 2010. DOI: `10.1109/IEEESTD.2010.5733835`.

[179] IEEE standard for software reviews and audits. (IEEE 1028-2008). DOI: `10.1109/IEEESTD.2008.4601584`.

[180] J. Iisakka and I. Tervonen. The darker side of inspection. In *Proceedings of Workshop on Inspection in Software Engineering (WISE 2001), Paris*, 2001.

[181] J. Iniesta. A tool and a set of metrics to support technical reviews, Nov. 1994. DOI: `10.2495/SQM940402`.

[182] P. Jaccard. The distribution of the flora in the alpine zone. *New phytologist*, 11(2):37–50, 1912.

[183] R. Jacob, A. Heinz, and J. P. Décieux. *Umfrage: Einführung in die Methoden der Umfrageforschung*. Walter de Gruyter, 2013.

[184] C. Jaspan, M. Jorde, A. Knight, C. Sadowski, E. K. Smith, C. Winter, and E. Murphy-Hill. Advantages and disadvantages of a monolithic repository: a case study at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pages 225–234. ACM, 2018.

[185] P. M. Johnson. An instrumented approach to improving software quality through formal technical review. In *Proceedings of the 16th International Conference on Software Engineering*, pages 113–122. IEEE Computer Society Press, 1994. DOI: `10.1109/ICSE.1994.296771`.

[186] P. M. Johnson. Supporting technology transfer of formal technical review through a computer supported collaborative review system. In *Proceedings of the Fourth International Conference on Software Quality*, 1994.

[187] P. M. Johnson and D. Tjahjono. Does every inspection really need a meeting? *Empirical Software Engineering*, 3(1):9–35, 1998.

[188] M. Kalinowski and G. H. Travassos. A computational framework for supporting software inspections. In *Automated Software Engineering, 2004. Proceedings. 19th International Conference on*, pages 46–55. IEEE, 2004.

[189] A. Kalyan, M. Chiam, J. Sun, and S. Manoharan. A collaborative code review platform for github. In *Engineering of Complex Computer Systems (ICECCS), 2016 21st International Conference on*, pages 191–196. IEEE, 2016. DOI: `10.1109/ICECCS.2016.032`.

[190] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. E. Hassan. Revisiting common bug prediction findings using effort-aware models. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.

[191] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2013.

[192] D. Kawrykow and M. P. Robillard. Non-essential changes in version histories. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 351–360. ACM, 2011.

[193] D. Kelly and T. Shepard. Task-directed software inspection. *Journal of Systems and Software*, 73(2):361–368, 2004.

[194] D. Kelly and T. Shepard. Task-directed software inspection technique: an experiment and case study. In *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*, page 6. IBM Press, 2000.

[195] L. P. W. Kim, C. Sauer, and R. Jeffery. A framework for software development technical reviews. In *Software Quality and Productivity*, pages 294–299. Springer, 1995.

[196] M. Kim, D. Notkin, D. Grossman, and G. Wilson. Identifying and summarizing systematic code changes via rule inference. *IEEE Transactions on Software Engineering*, 39(1):45–62, 2013.

[197] S. Kim, E. J. Whitehead Jr, and Y. Zhang. Classifying software changes: clean or buggy? *Software Engineering, IEEE Transactions on*, 34(2):181–196, 2008.

[198] S. Kim, H. Zhang, R. Wu, and L. Gong. Dealing with noise in defect prediction. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 481–490. IEEE, 2011.

[199] S. Kim, T. Zimmermann, K. Pan, E. James Jr, et al. Automatic identification of bug-introducing changes. In *21st IEEE/ACM International Conference on Automated Software Engineering*, pages 81–90. IEEE, 2006. DOI: 10.1109/ASE.2006.23.

[200] W. Kintsch, T. S. Mandel, and E. Kozminsky. Summarizing scrambled stories. *Memory & Cognition*, 5(5):547–552, 1977.

[201] J. C. Knight and E. A. Myers. Phased inspections and their implementation. *SIGSOFT Softw. Eng. Notes*, 16(3):29–35, July 1991. ISSN: 0163-5948. DOI: 10.1145/127099.127101.

[202] J. C. Knight and E. A. Myers. An improved inspection technique. *Communications of the ACM*, 36(11):50–61, 1993.

[203] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *Software Engineering, IEEE Transactions on*, 32(12):971–987, 2006.

[204] S. Kollanus and J. Koskinen. Software inspections in practice: six case studies. In *Product-Focused Software Process Improvement*, pages 377–382. Springer, 2006.

[205] M. Komssi, M. Kauppinen, M. Pyhajarvi, J. Talvio, and T. Mannisto. Persuading software development teams to document inspections: success factors and challenges in practice. In *Requirements Engineering Conference (RE), 2010 18th IEEE International*, pages 283–288. IEEE, 2010.

[206] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey. Investigating code review quality: do people and participation matter? In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 111–120. IEEE, 2015.

[207] O. Kononenko, T. Rose, O. Baysal, M. Godfrey, D. Theisen, and B. De Water. Studying pull request merges: a case study of shopify's active merchant. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pages 124–133. ACM, 2018. DOI: 10.1145/3183519.3183542.

[208] V. Kovalenko, N. Tintarev, E. Pasynkov, C. Bird, and A. Bacchelli. Does reviewer recommendation help developers? *IEEE Transactions on Software Engineering*, 2018. DOI: `10.1109/TSE.2018.2868367`.

[209] T. Krishnamurthy and S. Subramani. Ailments of distributed document reviews and remedies of doctor (document tree organizer tool) with distributed reviews support. In *2008 IEEE International Conference on Global Software Engineering*, pages 210–214, Aug. 2008. DOI: `10.1109/ICGSE.2008.8`.

[210] A. Kuhn, S. Ducasse, and T. Girba. Enriching reverse engineering with semantic clustering. In *12th Working Conference on Reverse Engineering (WCRE'05)*, 10–pp. IEEE, 2005.

[211] M. Kuhrmann, P. Diebold, J. Münch, P. Tell, V. Garousi, M. Felderer, K. Trektere, O. Linssen, E. Hanser, and C. R. Prause. Hybrid software and system development in practice: waterfall, scrum, and beyond. In *ICSSP 2017*, pages 30–39, 2017. DOI: `10.1145/3084100.3084104`.

[212] C. Ladas. *Scrumban-essays on kanban systems for lean software development*. Lulu.com, 2009. ISBN: 978-0578002149.

[213] J. S. Laguilles, E. A. Williams, and D. B. Saunders. Can lottery incentives boost web survey response rates? findings from four experiments. *Research in Higher Education*, 52(5):537–553, 2011.

[214] O. Laitenberger, S. Vegas, and M. Ciolkowski. The State of the Practice of Review and Review Technologies in Germany. Technical report, tech. report 011.02, Virtual Software Engineering Competence Center (VISEK), 2002.

[215] O. Laitenberger. *Cost-effective detection of software defects through perspective-based inspections*. PhD thesis, Universität Kaiserslautern, 2000.

[216] O. Laitenberger and J.-M. DeBaud. An encompassing life cycle centric survey of software inspection. *Journal of Systems and Software*, 50(1):5–31, 2000.

[217] O. Laitenberger and H. M. Dreyer. Evaluating the usefulness and the ease of use of a web-based inspection data collection tool. In *Software Metrics Symposium, 1998. Metrics 1998. Proceedings. Fifth International*, pages 122–132. IEEE, 1998. DOI: `10.1109/METRIC.1998.731237`.

[218] O. Laitenberger, M. Leszak, D. Stoll, and K. El Emam. Quantitative modeling of software reviews in an industrial setting. In *Software Metrics Symposium, 1999. Proceedings. Sixth International*, pages 312–322. IEEE, 1999.

[219] L. P. W. Land and J. Higgs. Factors contributing to software quality practices-an australian case study. In *Wireless Communications, Networking and Mobile Computing, 2007. WiCom 2007. International Conference on*, pages 5149–5152. IEEE, 2007.

[220] I. Langer, F. S. von Thun, and R. Tausch. *Sich verständlich ausdrücken*. E. Reinhardt, 9th edition, 2011.

[221] M. Lanna and D. Amyot. Spotting the difference. *Software: Practice and Experience*, 41(6):607–626, 2011.

[222] F. Lanubile and T. Mallardo. An empirical study of web-based inspection meetings. *Proceedings - 2003 International Symposium on Empirical Software Engineering, ISESE 2003*:244–251, Jan. 2003. DOI: `10.1109/ISESE.2003.1237984`.

[223] F. Lanubile and T. Mallardo. Preliminary evaluation of tool-based support for distributed inspection. In *Proceedings of ICSE International Workshop on Global Software Development*, Mar. 2002.

[224] F. Lanubile and T. Mallardo. Tool support for distributed inspection. In *2013 IEEE 37th Annual Computer Software and Applications Conference*, pages 1071–1071. IEEE Computer Society, 2002. DOI: 10.1109/CMPSAC.2002.1045151.

[225] F. Lanubile, T. Mallardo, and F. Calefato. Tool support for geographically dispersed inspection teams. *Software Process: Improvement and Practice*, 8(4):217–231, 2003. DOI: 10.1002/spip.184.

[226] F. Lanubile, T. Mallardo, F. Calefato, C. Denger, and M. Ciolkowski. Assessing the impact of active guidance for defect detection: a replicated experiment. In *Software Metrics, 2004. Proceedings. 10th International Symposium on*, pages 269–278. IEEE, 2004.

[227] T. D. LaToza, D. Garlan, J. D. Herbsleb, and B. A. Myers. Program comprehension as fact finding. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 361–370. ACM, 2007.

[228] A. M. Law and W. D. Kelton. *Simulation modeling and analysis.* McGraw-Hill, 2000.

[229] J. Lawrance, R. Bellamy, M. Burnett, and K. Rector. Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1323–1332. ACM, 2008.

[230] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming. How programmers debug, revisited: an information foraging theory perspective. *IEEE Transactions on Software Engineering*, 39(2):197–215, 2013.

[231] K. Lee and B. Boehm. Empirical results from an experiment on value-based review (vbr) processes. In *Empirical Software Engineering, 2005. 2005 International Symposium on*, 10–pp. IEEE, 2005.

[232] H. Leßmann. *Durchführung einer Umfrage-Studie zur Nutzung von Code Reviews in der Praxis.* Master's thesis, Leibniz Universität Hannover, 2017.

[233] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead Jr. Does bug prediction support human developers? findings from a google case study. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 372–381. IEEE Press, 2013.

[234] M. Li and S. Liu. Tool support for rigorous formal specification inspection. In *Computational Science and Engineering (CSE), 2014 IEEE 17th International Conference on*, pages 729–734. IEEE, 2014.

[235] M. Liro. *Computerunterstützung für systematische Literaturreviews mit der Snowballing-Technik.* Master's thesis, Leibniz Universität Hannover, 2016.

[236] M. Lumpe, R. Vasa, T. Menzies, R. Rush, and B. Turhan. Learning better inspection optimization policies. *International Journal of Software Engineering and Knowledge Engineering*, 22(05):621–644, 2012.

[237] F. Macdonald and J. Miller. A comparison of tool-based and paper-based software inspection. *Empirical Software Engineering*, 3(3):233–253, 1998. DOI: 10.1023/A:1009747104814.

[238] F. Macdonald and J. Miller. Modelling software inspection methods for the application of tool support. Technical report EFoCS-16-95, University of Strathclyde, 1995.

[239] F. Macdonald and J. Miller. A comparison of computer support systems for software inspection. *Automated Software Engineering*, 6(3):291–313, 1999. DOI: `10.1023/A:1008760911330`.

[240] F. Macdonald and J. Miller. A software inspection process definition language and prototype support tool. *Software Testing Verification and Reliability*:99–128, Jan. 1997. DOI: `10.1002/(SICI)1099-1689(199706)7:2<99::AID-STVR133>3.0.CO;2-Q`.

[241] L. MacLeod, M. Greiler, M.-A. Storey, C. Bird, and J. Czerwonka. Code reviewing in the trenches: understanding challenges and best practices. *IEEE Software*, 35(4):34–42, 2017. DOI: `10.1109/MS.2017.265100500`.

[242] R. J. Madachy. System dynamics modeling of an inspection-based process. In *Software Engineering, 1996., Proceedings of the 18th International Conference on*, pages 376–386, Berlin, Germany. IEEE, 1996.

[243] R. J. Madachy. *Software Process Dynamics*. Wiley, 2007. ISBN: 9780470192702.

[244] J. T. Madhavan and E. J. Whitehead Jr. Predicting buggy changes inside an integrated development environment. In *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, pages 36–40. ACM, 2007.

[245] J. I. Maletic and A. Marcus. Using latent semantic analysis to identify similarities in source code to support program understanding. In *Tools with Artificial Intelligence, 2000. ICTAI 2000. Proceedings. 12th IEEE International Conference on*, pages 46–53. IEEE, 2000.

[246] R. Malhotra. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing*, 27:504–518, 2015.

[247] S. Mancoridis, B. S. Mitchell, C. Rorres, Y.-F. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *IWPC*, volume 98, pages 45–52, 1998.

[248] M. V. Mantyla and C. Lassenius. What types of defects are really discovered in code reviews? *Software Engineering, IEEE Transactions on*, 35(3):430–448, 2009.

[249] J. G. March and H. A. Simon. *Organizations*. John Wiley & Sons, Inc., 1958.

[250] G. Mariscal, O. Marban, and C. Fernandez. A survey of data mining and knowledge discovery process models and methodologies. *The Knowledge Engineering Review*, 25(2):137–166, 2010.

[251] V. Mashayekhi, J. M. Drake, W.-T. Tsal, and J. Riedl. Distributed, collaborative software inspection. *IEEE Software*:66–75, Jan. 1993. DOI: `10.1109/52.232404`.

[252] V. Mashayekhi, C. Feulner, and J. Riedl. Cais: collaborative asynchronous inspection of software. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 21–34. Association for Computing Machinery, Dec. 1994. DOI: `10.1145/193173.195290`.

[253] J. Matsuda, S. Hayashi, and M. Saeki. Hierarchical categorization of edit operations for separately committing large refactoring results. In *Proceedings of the 14th International Workshop on Principles of Software Evolution*, pages 19–27. ACM, 2015.

[254] T. Matsumura, A. Monden, and K.-i. Matsumoto. The detection of faulty code violating implicit coding rules. In *Empirical Software Engineering, 2002. Proceedings. 2002 International Symposium on*, pages 173–182. IEEE, 2002.

[255] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, (4):308–320, 1976. DOI: 10.1109/TSE.1976.233837.

[256] S. McIntosh and Y. Kamei. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. *IEEE Transactions on Software Engineering*, 44:412–428, 2017. DOI: 10.1109/TSE.2017.2693980.

[257] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21(5):2146–2189, 2015. DOI: 10.1007/s10664-015-9381-9.

[258] D. A. McMeekin, B. R. von Konsky, E. Chang, and D. J. Cooper. Evaluating software inspection cognition levels using bloom's taxonomy. In *Software Engineering Education and Training, 2009. CSEET'09. 22nd Conference on*, pages 232–239. IEEE, 2009.

[259] D. A. McMeekin, B. R. von Konsky, M. Robey, and D. J. Cooper. The significance of participant experience when evaluating software inspection techniques. In *Software Engineering Conference, 2009. ASWEC'09. Australian*, pages 200–209. IEEE, 2009.

[260] A. McNair, D. M. German, and J. Weber-Jahnke. Visualizing software architecture evolution using change-sets. In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pages 130–139. IEEE, 2007.

[261] D. S. McNamara, E. Kintsch, N. B. Songer, and W. Kintsch. Are good texts always better? interactions of text coherence, background knowledge, and levels of understanding in learning from text. *Cognition and instruction*, 14(1):1–43, 1996.

[262] M. Melis, I. Turnu, A. Cau, and G. Concas. Evaluating the impact of test-first programming and pair programming through software process simulation. *Software Process: Improvement and Practice*, 11(4):345–360, 2006.

[263] A. Meneely, L. Williams, W. Snipes, and J. Osborne. Predicting failures with developer networks and social network analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 13–23. ACM, 2008.

[264] T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald. Problems with precision: a response to "comments on 'data mining static code attributes to learn defect predictors'". *IEEE Transactions on Software Engineering*, 33(9):637–640, 2007.

[265] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, 17(4):375–407, 2010.

[266] B. Meyer. Design and code reviews in the age of the internet. In *Software Engineering Approaches for Offshore and Outsourced Development*, pages 126–133. Springer, 2008.

[267] B. J. Meyer. Reading research and the composition teacher: the importance of plans. *College composition and communication*:37–49, 1982.

[268] G. A. Miller. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological review*, 63(2):81, 1956.

[269] J. Miller, J. D. Ferguson, and P. Murphy. Groupware support for asynchronous document review. In *Proceedings of the 17th Annual International Conference on Computer Documentation*, pages 185–192. ACM, 1999. DOI: 10.1145/318372.318592.

[270] J. Miller and F. MacDonald. Empirical incremental approach to tool evaluation and improvement. *Journal of Systems and Software*:19–35, Apr. 2000. DOI: 10.1016/S0164-1212(99)00107-7.

[271] J. Miller, F. Macdonald, and J. Ferguson. Assisting management decisions in the software inspection process. *Information Technology and Management*, 3(1-2):67–83, 2002. DOI: 10.1023/A:1013112826330.

[272] A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.

[273] J. Moeyersoms, E. J. de Fortuny, K. Dejaeger, B. Baesens, and D. Martens. Comprehensible software fault and effort prediction: a data mining approach. *Journal of Systems and Software*, 100:80–90, 2015.

[274] M. I. Mukadam. *git-reviewed: A Distributed Peer Review Tool & User Study*. Master's thesis, Concordia University, 2014.

[275] M. Mukadam, C. Bird, and P. C. Rigby. Gerrit software code review data from android. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 45–48. IEEE, 2013.

[276] J. Münch and O. Armbrust. Using empirical knowledge from replicated experiments for software process simulation: a practical example. In *Empirical Software Engineering, 2003. ISESE 2003. Proceedings. 2003 International Symposium on*, pages 18–27, Rome, Italy. IEEE, 2003.

[277] E. Mustonen-Ollila and K. Lyytinen. Why organizations adopt information system process innovations: a longitudinal study using diffusion of innovation theory. *Information Systems Journal*, 13(3):275–297, 2003.

[278] B. A. Myers, A. J. Ko, T. D. LaToza, and Y. Yoon. Programmers are users too: human-centered methods for improving programming tools. *Computer*, 49(7):44–52, 2016.

[279] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 284–292. IEEE, 2005.

[280] M. Nayrolles and A. Hamou-Lhadj. Clever: combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects. In *MSR '18 Proceedings of the 15th International Conference on Mining Software Repositories*, pages 153–164, 2018. DOI: 10.1145/3196398.3196438.

[281] H. Neu, T. Hanne, J. Münch, S. Nickel, and A. Wirsen. Creating a code inspection model for simulation-based decision support. In *ProSim*, volume 3, pages 3–4, 2003.

[282] M. Nick, C. Denger, and T. Willrich. Experience-based support for code inspections. In *Professional Knowledge Management*, pages 121–126. Springer, 2005.

[283] A. Nwesri and K. Hashim. A model and tool features for collaborative artifact inspection and review. *WSEAS Transactions on Systems*:1038–1047, Oct. 2008.

[284] M. P. O'Brien and J. Buckley. Inference-based and expectation-based processing in program comprehension. In *Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on*, pages 71–78. IEEE, 2001.

[285] C. Oezbek and L. Prechelt. Jtourbus: simplifying program understanding by documentation that provides tours through the source code. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 64–73. IEEE, 2007.

[286] W. J. Orlikowski. Case tools as organizational change: investigating incremental and radical changes in systems development. *MIS quarterly*:309–340, 1993. DOI: 10.2307/249774.

[287] H. Osman and O. M. Nierstrasz. *Empirically-Grounded Construction of Bug Prediction and Detection Tools*. PhD thesis, Universität Bern, 2017.

[288] F. L. Oswald, S. T. McAbee, T. S. Redick, and D. Z. Hambrick. The development of a short domain-general measure of working memory capacity. *Behavior research methods*, 47(4):1343–1355, 2015.

[289] F. G. Paas and J. J. Van Merriënboer. Instructional control of cognitive load in the training of complex cognitive tasks. *Educational psychology review*, 6(4):351–371, 1994.

[290] F. Padberg, T. Ragg, and R. Schoknecht. Using machine learning for estimating the defect content after an inspection. *IEEE Transactions on Software Engineering*, 30:17–28, 2004.

[291] B. Page and W. Kreuzer. *The Java Simulation Handbook – Simulating Discrete Event Systems with UML and Java*. Shaker, 2005.

[292] S. Panichella, V. Arnaoudova, M. Di Penta, and G. Antoniol. Would static analysis tools help developers with code reviews? In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 161–170. IEEE, 2015.

[293] L. Pascarella, F. Palomba, and A. Bacchelli. Re-evaluating method-level bug prediction. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 592–601. IEEE, 2018.

[294] K. Pawlikowski, H.-D. Jeong, and J.-S. Lee. On credibility of simulation studies of telecommunication networks. *Communications Magazine, IEEE*, 40(1):132–139, 2002.

[295] J. Pearl. *Causality: models, reasoning, and inference*. Cambridge University Press, 2001.

[296] J. M. Perpich, D. E. Perry, A. A. Porter, L. G. Votta, and M. W. Wade. Anywhere, anytime code inspections: using the web to remove inspection bottlenecks in large-scale software development. In *Proceedings of the 19th International Conference on Software Engineering*, pages 14–21. ACM, 1997. DOI: 10.1145/253228.253234.

[297] D. E. Perry, A. Porter, M. W. Wade, L. G. Votta, and J. Perpich. Reducing inspection interval in large-scale software development. *Software Engineering, IEEE Transactions on*, 28(7):695–705, 2002. DOI: 10.1109/TSE.2002.1019483.

[298] K. Petersen, K. Rönkkö, and C. Wohlin. The impact of time controlled reading on software inspection effectiveness and efficiency: a controlled experiment. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 139–148. ACM, 2008.

[299] S. Platz, M. Taeumel, B. Steinert, R. Hirschfeld, and H. Masuhara. Unravel programming sessions with thresher: identifying coherent and complete sets of fine-granular source code changes. In *Proceedings of the 32nd JSSST Annual Conference*, pages 24–39, 2016. DOI: `10.11185/imt.12.24`.

[300] A. A. Porter, H. P. Siy, C. A. Toman, and L. G. Votta. An experiment to assess the cost-benefits of code inspections in large scale software development. *IEEE transactions on software engineering*, 23(6):329–346, 1997. DOI: `10.1109/32.601071`.

[301] A. Porter, H. Siy, A. Mockus, and L. Votta. Understanding the sources of variation in software inspections. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(1):41–79, 1998.

[302] A. Porter, H. Siy, and L. Votta. A review of software inspections. *Advances in Computers*, 42:39–76, 1996.

[303] H. Potelle and J.-F. Rouet. Effects of content representation and readers' prior knowledge on the comprehension of hypertext. *International Journal of Human-Computer Studies*, 58(3):327–345, 2003.

[304] C. R. Prause and S. Apelt. An approach for continuous inspection of source code. In *Proceedings of the 6th international workshop on Software quality*, pages 17–22. ACM, 2008.

[305] C. R. Prause and M. Eisenhauer. Social aspects of a continuous inspection platform for software source code. In *Proceedings of the 2008 international workshop on Cooperative and human aspects of software engineering*, pages 85–88. ACM, 2008.

[306] R. Priest and B. Plimmer. Rca: experiences with an ide annotation tool. In *Proceedings of the 7th ACM SIGCHI New Zealand Chapter's International Conference on Computer-human Interaction: Design Centered HCI*, pages 53–60. ACM, 2006. DOI: `10.1145/1152760.1152767`.

[307] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2014. URL: `https://www.R-project.org`.

[308] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2017. URL: `https://www.R-project.org/`.

[309] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič. Software fault prediction metrics: a systematic literature review. *Information and Software Technology*, 55(8):1397–1418, 2013.

[310] M. M. Rahman, C. K. Roy, J. Redl, and J. A. Collins. Correct: code reviewer recommendation at github for vendasta technologies. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, pages 792–797. IEEE, 2016. DOI: `10.1145/2970276.2970283`.

[311] J. Rasmussen. Skills, rules, and knowledge; signals, signs, and symbols, and other distinctions in human performance models. *IEEE transactions on systems, man, and cybernetics*, SMC-13(3):257–266, 1983. DOI: `10.1109/TSMC.1983.6313160`.

[312] J. Ratcliffe. Moving software quality upstream: the positive impact of lightweight peer code review. In *Pacific NW Software Quality Conference*, 2009.

[313] J. Ratzinger, M. Pinzger, and H. Gall. Eq-mine: predicting short-term defects for software evolution. In *Fundamental Approaches to Software Engineering*, pages 12–26. Springer, 2007.

[314] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu. On the "naturalness" of buggy code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 428–439, May 2016. DOI: 10.1145/2884781.2884848.

[315] T. Raz and A. T. Yaung. Factors affecting design inspection effectiveness in software development. *Information and Software Technology*, 39(4):297–305, 1997.

[316] S. Rifkin and L. Deimel. Applying program comprehension techniques to improve software inspections. In *Proceedings of the 19th annual NASA software engineering laboratory workshop, Greenbelt, MD*, 1994.

[317] P. C. Rigby. *Understanding open source software peer review: Review processes, parameters and statistical models, and underlying behaviours and mechanisms.* PhD thesis, University of Victoria, 2011.

[318] P. C. Rigby and C. Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 202–212, Saint Petersburg, Russia. ACM, 2013.

[319] P. C. Rigby, B. Cleary, F. Painchaud, M. Storey, and D. M. German. Contemporary peer review in action: lessons from open source development. *Software, IEEE*, 29(6):56–61, 2012.

[320] P. C. Rigby, D. M. German, L. Cowen, and M.-A. Storey. Peer review on open source software projects: parameters, statistical models, and theory. *ACM Transactions on Software Engineering and Methodology*, 23:35:1–35:33, 2014. DOI: 10.1145/2594458.

[321] P. C. Rigby and M.-A. Storey. Understanding broadcast based peer review on open source software projects. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 541–550. ACM, 2011.

[322] B. Robbins and J. Carver. Cognitive factors in perspective-based reading (pbr): a protocol analysis study. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 145–155. IEEE Computer Society, 2009.

[323] M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: an exploratory study. *Software Engineering, IEEE Transactions on*, 30(12):889–903, 2004.

[324] E. M. Rogers. *Diffusion of Innovations*. Free Press, 5th edition, 2003.

[325] M. Roper, M. Wood, and J. Miller. An empirical evaluation of defect detection techniques. *Information and Software Technology*, 39(11):763–775, 1997.

[326] C. Rosen, B. Grawi, and E. Shihab. Commit guru: analytics and risk prediction of software commits. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 966–969. ACM, 2015.

[327] C. Rudin. Please stop explaining black box models for high stakes decisions. *arXiv preprint arXiv:1811.10154*, 2018.

[328] P. Runeson, M. Höst, A. Rainer, and B. Regnell. *Case Study Resarch in Software Engineering – Guidelines and Examples*. Wiley, 2012.

[329] I. Rus, M. Halling, and S. Biffl. Supporting decision-making in software engineering with process simulation and empirical studies. *International Journal of Software Engineering and Knowledge Engineering*, 13(05):531–545, 2003.

[330] D. Ryu and J. Baik. Effective multi-objective naïve bayes learning for cross-project defect prediction. *Applied Soft Computing*, 49:1062–1077, 2016.

[331] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli. Modern code review: a case study at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pages 181–190. ACM, 2018. DOI: 10.1145/3183519.3183525.

[332] A. Saltelli, K. Chan, and E. M. Scott, editors. *Sensitivity Analysis*. John Wiley & Sons, Inc., 2000.

[333] M.-L. Sánchez-Gordón and R. V. O'Connor. Understanding the gap between software process practices and actual practice in very small companies. *Software Quality Journal*:1–22, 2015.

[334] R. G. Sargent. Verification and validation of simulation models. In *Proceedings of the 37th conference on Winter simulation*, pages 130–143. winter simulation conference, 2005.

[335] T. D. Sasso, A. Mocci, M. Lanza, and E. Mastrodicasa. How to gamify software engineering. In *SANER 2017, Proceedings of*, 2017. DOI: 10.1109/SANER.2017.7884627.

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

[337] K. Schneider. Prototypes as assets, not toys: why and how to extract knowledge from prototypes. In *Proceedings of the 18th International Conference on Software Engineering*, pages 522–531. IEEE Computer Society, 1996.

[338] V. Sembugamoorthy and L. Brothers. Icicle: intelligent code inspection in a c language environment. In *Proceedings - IEEE Computer Society's International Computer Software & Applications Conference*, pages 146–154. IEEE, Piscataway, NJ, Dec. 1990. DOI: 10.1109/CMPSAC.1990.139343.

[339] E. Shihab. *An exploration of challenges limiting pragmatic software defect prediction.* PhD thesis, Queen's University (Canada), 2012.

[340] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang. An industrial study on the risk of software changes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 62. ACM, 2012.

[341] E. Shihab, A. Mockus, Y. Kamei, B. Adams, and A. E. Hassan. High-impact defects: a study of breakage and surprise defects. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 300–310. ACM, 2011.

[342] J. Shimagaki, J. Shimagaki, Y. Kamei, S. McIntosh, A. E. Hassan, and N. Ubayashi. A study of the quality-impacting practices of modern code review at sony mobile. In *ICSE '16 Companion*, 2016. DOI: 10.1145/2889160.2889243.

[343] Y. Shin, R. Bell, T. Ostrand, and E. Weyuker. Does calling structure information improve the accuracy of fault prediction? In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*, pages 61–70. IEEE, 2009.

255

[344] T. J. Shippey. *Exploiting Abstract Syntax Trees to Locate Software Defects*. PhD thesis, University of Hertfordshire, 2015.

[345] S. Shivaji, E. J. Whitehead, R. Akella, and S. Kim. Reducing features to improve code change-based bug prediction. *IEEE Transactions on Software Engineering*, 39(4):552–569, 2013.

[346] J. Siedersleben. *Moderne Softwarearchitektur*. dpunkt.verlag, 2004.

[347] J. Siegmund, C. Kästner, S. Apel, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann. Understanding understanding source code with functional magnetic resonance imaging. In *Proceedings of the 36th International Conference on Software Engineering*, pages 378–389. ACM, 2014.

[348] H. A. Simon. How big is a chunk? *Science*, 183(4124):482–488, 1974.

[349] E. Singer and C. Ye. The use and effects of incentives in surveys. *The ANNALS of the American Academy of Political and Social Science*, 645(1):112–141, 2013.

[350] D. I. Sjøberg, T. Dybå, B. C. Anda, and J. E. Hannay. Building theories in software engineering. In *Guide to advanced empirical software engineering*, pages 312–336. Springer, 2008.

[351] D. I. Sjøberg, J. E. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanovic, N.-K. Liborg, and A. C. Rekdal. A survey of controlled experiments in software engineering. *Software Engineering, IEEE Transactions on*, 31(9):733–753, 2005.

[352] M. Skoglund and V. Kjellgren. An experimental comparison of the effectiveness and usefulness of inspection techniques for object-oriented programs. In *8th International Conference on Empirical Assessment in Software Engineering (EASE 2004)*, pages 165–174. IET, 2004. DOI: `10.1049/ic:20040409`.

[353] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *ACM sigsoft software engineering notes*, 30(4):1–5, 2005.

[354] SmartBear. The state of code quality 2016. URL: `https://smartbear.com/resources/ebooks/state-of-code-quality-2016/`. last checked 2017-06-13.

[355] B. Soltanifar, A. Erdem, and A. Bener. Predicting defectiveness of software patches. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 22. ACM, 2016. DOI: `10.1145/2961111.2962601`.

[356] D. Spadini, F. Palomba, T. Baum, S. Hanenberg, M. Bruntink, and A. Bacchelli. Test-driven code review: an empirical study. In *Software Engineering (ICSE), 2019 41st International Conference on*, 2019.

[357] D. Spencer. Card sorting: a definitive guide. http://boxesandarrows.com/card-sorting-a-definitive-guide/, 2004.

[358] M. V. Stein, M. P. E. Heimdahl, and J. T. Riedl. Enhancing annotation visibility for software inspection. In *14th IEEE International Conference on Automated Software Engineering*, pages 243–246, Aug. 1999. DOI: `10.1109/ASE.1999.802288`.

[359] M. Stein, J. Riedl, S. J. Harner, and V. Mashayekhi. A case study of distributed, asynchronous software inspection. In *Proceedings of the 19th International Conference on Software Engineering*, pages 107–117. ACM, 1997. DOI: `10.1145/253228.253250`.

[360] B. Steinert, M. Taeumel, J. Lincke, T. Pape, and R. Hirschfeld. Codetalk conversations about code. In *2010 Eighth International Conference on Creating, Connecting and Collaborating through Computing*, pages 11–18, Jan. 2010. DOI: 10.1109/C5.2010.11.

[361] M.-A. Storey, F. D. Fracchia, and H. A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Systems and Software*, 44(3):171–185, 1999.

[362] M.-A. Storey. Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Journal*, 14(3):187–208, 2006.

[363] S. Sudman and N. M. Bradburn. *Asking questions: a practical guide to questionnaire design.* San Francisco Calif. Jossey-Bass Publishers, 1982.

[364] J. Sweller. Cognitive load during problem solving: effects on learning. *Cognitive science*, 12(2):257–285, 1988.

[365] M. Tan, L. Tan, S. Dara, and C. Mayeux. Online defect prediction for imbalanced data. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 99–108. IEEE Press, 2015.

[366] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering*, 2018. DOI: 10.1109/TSE.2018.2794977.

[367] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim. How do software engineers understand code changes?: an exploratory study in industry. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012.

[368] Y. Tao and S. Kim. Partitioning composite code changes to facilitate code review. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 180–190. IEEE, 2015.

[369] A. Tarvo, N. Nagappan, T. Zimmermann, T. Bhat, and J. Czerwonka. Predicting risk of pre-release code changes with checkinmentor. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, 2013. DOI: 10.1109/ISSRE.2013.6698912.

[370] I. Tervonen, L. Harjumaa, and J. Iisakka. The virtual logging meeting: a web-based solution to resource problems in software inspection. In *Proceedings of the Sixth European Conference on Software Quality*, pages 342–351. Citeseer, 1999.

[371] S. Thangthumachit, S. Hayashi, and M. Saeki. Understanding source code differences by separating refactoring effects. In *Software Engineering Conference (APSEC), 2011 18th Asia Pacific*, pages 339–347. IEEE, 2011.

[372] T. Thelin, P. Andersson, and J. Harrell. Tool support for usage-based reading. In *IASTED Conf. on Software Engineering*, pages 601–606, 2004. ISBN: 0-88986-410-1.

[373] T. Thelin, P. Runeson, and B. Regnell. Usage-based reading—an experiment to guide reviewers with use cases. *Information and Software Technology*, 43(15):925–938, 2001.

[374] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida. Investigating code review practices in defective files: an empirical study of the qt system. In *MSR '15 Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 168–179, 2015.

[375] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *Proceedings of the 38th international conference on software engineering*, pages 1039–1050. ACM, 2016. DOI: 10.1145/2884781.2884852.

[376] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K.-i. Matsumoto. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 141–150, 2015. DOI: 10.1109/SANER.2015.7081824.

[377] S. R. Tilley, S. Paul, and D. B. Smith. Towards a framework for program understanding. In *Program Comprehension, 1996, Proceedings., Fourth Workshop on*, pages 19–28. IEEE, 1996.

[378] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili. Detecting defects in object-oriented designs: using reading techniques to increase software quality. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 47–56, Denver, Colorado, USA. ACM, 1999. ISBN: 1-58113-238-7. DOI: 10.1145/320384.320389. URL: http://doi.acm.org/10.1145/320384.320389.

[379] R. H. Trigg. Guided tours and tabletops: tools for communicating in a hypertext environment. *ACM Transactions on Information Systems (TOIS)*, 6(4):398–414, 1988.

[380] I. Turnu, M. Melis, A. Cau, A. Setzu, G. Concas, and K. Mannaro. Modeling and simulation of open source development using an agile practice. *Journal of Systems Architecture*, 52(11):610–618, 2006.

[381] J. D. Tvedt and J. Collofello. Evaluating the effectiveness of process improvements on software development cycle time via system dynamics modelling. In *Computer Software and Applications Conference, 1995. COMPSAC 95. Proceedings., Nineteenth Annual International*, pages 318–325, Dallas, TX, USA. IEEE, 1995.

[382] A. Tversky and D. Kahneman. Judgment under uncertainty: heuristics and biases. In *Utility, probability, and human decision making*, pages 141–162. Springer, 1975.

[383] V. Tzerpos and R. C. Holt. Acdc: an algorithm for comprehension-driven clustering. In *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*, pages 258–267. IEEE, 2000.

[384] N. Unsworth, R. P. Heitz, J. C. Schrock, and R. W. Engle. An automated version of the operation span task. *Behavior research methods*, 37(3):498–505, 2005.

[385] M. Van Genuchten, W. Cornelissen, and C. Van Dijk. Supporting inspections with an electronic meeting system. *Journal of Management Information Systems*:165–178, Jan. 1997. DOI: 10.1080/07421222.1997.11518179.

[386] O. Vandecruys, D. Martens, B. Baesens, C. Mues, M. De Backer, and R. Haesen. Mining software repositories for comprehensible software fault prediction models. *Journal of Systems and software*, 81(5):823–839, 2008.

[387] A. M. Vans, A. von Mayrhauser, and G. Somlo. Program understanding behavior during corrective maintenance of large-scale software. *International Journal of Human-Computer Studies*, 51:31–70, 1999.

[388] W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S*. Springer, New York, fourth edition, 2002. URL: http://www.stats.ox.ac.uk/pub/MASS4. ISBN 0-387-95457-0.

[389] A. von Mayrhauser and A. M. Vans. Industrial experience with an integrated code comprehension model. *Software Engineering Journal*, 10(5):171–182, 1995.

[390] L. G. Votta. Does every inspection need a meeting? *ACM SIGSOFT Software Engineering Notes*, 18(5):107–114, 1993.

[391] W. W. Wakeland, R. H. Martin, and D. Raffo. Using design of experiments, sensitivity analysis, and hybrid simulation to evaluate changes to a software development process: a case study. *Software Process: Improvement and Practice*, 9(2):107–119, 2004.

[392] A. Walenstein. *Cognitive support in software engineering tools: A distributed cognition framework*. PhD thesis, Simon Fraser University, 2002.

[393] A. Walenstein. Observing and measuring cognitive support: steps toward systematic tool evaluation and engineering. In *Program Comprehension, 2003. 11th IEEE International Workshop on*, pages 185–194. IEEE, 2003.

[394] A. Walenstein. Theory-based analysis of cognitive support in software comprehension tools. In *Program Comprehension, 2002. Proceedings. 10th International Workshop on*, pages 75–84. IEEE, 2002.

[395] J. Wang, P. C. Shih, Y. Wu, and J. M. Carroll. Comparative case studies of open source software peer review practices. *Information and Software Technology*, 67:1–12, 2015.

[396] R. Wen, D. Gilbert, M. G. Roche, and S. McIntosh. Blimp tracer: integrating build impact analysis with code review. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 685–694. IEEE, 2018. DOI: 10.1109/ICSME.2018.00078.

[397] K. Wiegers. *Peer reviews in software: a practical guide*. Addison-Wesley information technology series. Addison-Wesley, 2002. ISBN: 9780201734850.

[398] R. R. Wilcox. *Introduction to robust estimation and hypothesis testing*. Academic press, 2011.

[399] O. Wilhelm, A. Hildebrandt, and K. Oberauer. What is working memory capacity, and how can we measure it? *Frontiers in psychology*, 4, 2013. DOI: 10.3389/fpsyg.2013.00433.

[400] C. C. Williams and J. K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering*, 31:466–480, 2005.

[401] D. Winkler, S. Biffl, and K. Faderl. Investigating the temporal behavior of defect detection in software inspection and inspection-based testing. In *International Conference on Product Focused Software Process Improvement*, pages 17–31. Springer, 2010.

[402] D. Winkler, S. Biffl, and B. Thurnher. Investigating the impact of active guidance on design inspection. In F. Bomarius and S. Komi-Sirviö, editors, *Product Focused Software Process Improvement*, pages 458–473, Berlin, Heidelberg. Springer Berlin Heidelberg, 2005. ISBN: 978-3-540-31640-4. DOI: 10.1007/11497455_36.

[403] M. Winter, K. Vosseberg, and A. Spillner. *Umfrage 2016 "Softwaretest in Praxis und Forschung"*. dpunkt.verlag, 2016.

[404] I. H. Witten, E. Frank, and M. A. Hall. *Data Mining – Practical Machine Learning Tools and Techniques*. Elsevier, 3rd edition, 2011.

[405] C. Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *EASE '14 Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, 2014. DOI: 10.1145/2601248.2601268.

[406] X. Xia, D. Lo, X. Wang, and X. Yang. Collective personalized change classification with multiobjective search. *IEEE Transactions on Reliability*, 65(4):1810–1829, 2016.

[407] T. Yamashita. *Evaluation of Jupiter, a Lightweight Code Review Framework*. Master's thesis, University of Hawai'i, 2006.

[408] R. A. Yaros. Effects of text and hypertext structures on user interest and understanding of science and technology. *Science Communication*, 33(3):275–308, 2011.

[409] R. A. Yaros. Is it the medium or the message? structuring complex news to enhance engagement and situational understanding by nonexperts. *Communication Research*, 33(4):285–309, 2006.

[410] Y. Yu, T. T. Tun, and B. Nuseibeh. Specifying and detecting meaningful changes in programs. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 273–282. IEEE Computer Society, 2011.

[411] H. Zhang, B. Kitchenham, and D. Pfahl. Software process simulation modeling: an extended systematic review. In *New Modeling Concepts for Today's Software Processes*, pages 309–320. Springer, 2010.

[412] Q. Zhang. *Improving software development process and project management with software project telemetry*. PhD thesis, University of Hawai'i, 2006.

[413] T. Zhang, M. Song, J. Pinedo, and M. Kim. Interactive code review for systematic changes. In *Proceedings of 37th IEEE/ACM International Conference on Software Engineering. IEEE*, pages 111–122, 2015. DOI: 10.1109/ICSE.2015.33.

[414] Y.-M. Zhu. *Software Reading Techniques: Twenty Techniques for More Effective Software Review and Inspection*. apress, 2016.

Glossary

API Application Programming Interface.

Author In the context of a code review, the author is the developer that implemented the unit of work to be reviewed.

BIC Bayesian Information Criterion.

CAQDAS Computer-Aided Qualitative Data Analysis.

CI Continuous Integration.

CLI Command Line Interface.

Change Part The elements of a code change are called 'change parts'. In its simplest form, a change part corresponds directly to a change hunk as given by the Unix diff tool or the version control system. In the context of Chapters 13 and 14, we combined hunks from the same method into one change part.

Changeset Used as synonym to 'Code Change' in this thesis.

Code Change The 'code change' consists of all changes to source files performed in the unit of work under review (see also Definition 2 on page 24). The code change defines the scope of the review, i.e., the parts of the code base that shall be reviewed.

Code Review See Definition 1 on page 23.

Cognitive Load 'Cognitive load' is a multidimensional construct that represents the load that performing a particular task imposes on the human cognitive system. It depends on traits of the task, of the environment, of the human (e.g., the working memory capacity) and the mental effort spent. [289]

Delocalized Defect A defect that can only be found or found much more easily by combining knowledge about several parts of the source code. [109]

IDE Integrated Development Environment.

Mental Load The term 'mental load' of a task is used to refer to the subset of factors that influence the task's cognitive load which depends only on the task or environment, i.e., which is independent of subject characteristics. [289]

QA Quality Assurance.

Regular, Change-Based Code Review See Definition 2 on page 24.

Review Effectiveness 'Review effectiveness' is the ratio of defects found to all defects in the code change. [52]

Review Efficiency 'Review efficiency' is the number of defects found per review hour invested. [52]

Reviewer The humans performing the code review, excluding the author, are called 'reviewers' (see also Definition 1 on page 23).

SCM Source Code Management system, e.g., Subversion or git.

SE Software Engineering.

Tour A 'tour' is a sequence (permutation) of all change parts of a code change. This thesis also uses 'code change part order' as a synonym.

Trigger (for review remark) A trigger for a review remarks is a change part that, when reviewed, leads to the creation of that remark. The term is mainly used in the context of Chapter 15's data mining study.

UI User Interface.

UML Unified Modeling Language.

Working Memory 'Working memory' is the part of human memory that is needed for short-term storage during information processing. Its capacity can be measured using 'complex span tests'. [399]

List of Figures

List of Tables

267

List of Definitions

Curriculum Vitae

Personal Details

Name:	Tobias Baum
Date of Birth:	28.04.1985
City of Birth:	Hannover, Germany

Education

1991 – 1995	Friedrich-Ebert-Grundschule, Hannover
1995 – 1997	Orientierungsstufe Badenstedt, Hannover
1997 – 2004	Helene-Lange-Schule, Hannover, Abitur
2004 – 2007	FHDW Hannover, Studium Informatik, Diplom-Informatiker (FH)
2007 – 2009	FHDW Hannover, Studium Business Process Engineering, Master of Engineering
2014 – 2019	Leibniz Universität Hannover, Promotion Informatik

Professional Experience

2000 – 2004	SET GmbH, Hannover, side job
2004 – 2007	SET GmbH, Hannover, working student
2007 – 2009	SET GmbH, Hannover, developer
2009 – 2018	SET GmbH, Hannover, lead developer
2019 ongoing	SET GmbH, Hannover, director

Peer-Reviewed Publications

- SPADINI, D., PALOMBA, F., BAUM, T., HANENBERG, S., BRUNTINK, M., AND BACCHELLI, A. Test-driven code review: An empirical study. In *Software Engineering (ICSE), 2019 41st International Conference on* (2019). [356]
- FREGNAN, E., BAUM, T., PALOMBA, F., AND BACCHELLI, A. A survey on software coupling relations and tools. *Information and Software Technology 107* (2019), 159 – 178. [130]
- BAUM, T., BACCHELLI, A., AND SCHNEIDER, K. Associating working memory capacity and code change ordering with code review performance. *Empirical Software Engineering* (2018). [29]

- BAUM, T., LESSMANN, H., AND SCHNEIDER, K. The choice of code review process: A survey on the state of the practice. In *Product-Focused Software Process Improvement* (Cham, 2017), M. Felderer, D. Méndez Fernández, B. Turhan, M. Kalinowski, F. Sarro, and D. Winkler, Eds., Springer International Publishing, pp. 111–127. [37]
- BAUM, T., SCHNEIDER, K., AND BACCHELLI, A. On the optimal order of reading source code changes for review. In *33rd IEEE International Conference on Software Maintenance and Evolution (ICSME), Proceedings* (2017), pp. 329–340. [41]
- BAUM, T., KORTUM, F., SCHNEIDER, K., BRACK, A., AND SCHAUDER, J. Comparing pre-commit reviews and post-commit reviews using process simulation. *Journal of Software: Evolution and Process 29*, 11 (2017), e1865. [35][1]
- BAUM, T., AND SCHNEIDER, K. On the need for a new generation of code review tools. In *Product-Focused Software Process Improvement: 17th International Conference, PROFES 2016, Trondheim, Norway, November 22-24, 2016, Proceedings 17* (2016), Springer, pp. 301–308. [40]
- BAUM, T., LISKIN, O., NIKLAS, K., AND SCHNEIDER, K. Factors influencing code review processes in industry. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2016), FSE 2016, ACM, pp. 85–96. [39]
- BAUM, T., LISKIN, O., NIKLAS, K., AND SCHNEIDER, K. A faceted classification scheme for change-based industrial code review processes. In *Software Quality, Reliability and Security (QRS), 2016 IEEE International Conference on* (Vienna, Austria, 2016), IEEE, pp. 74–85. [38]
- BAUM, T., KORTUM, F., SCHNEIDER, K., BRACK, A., AND SCHAUDER, J. Comparing pre commit reviews and post commit reviews using process simulation. In *Software and System Process (ICSSP), 2016 International Conference on* (Austin, TX, USA, 2016). [34]
- BAUM, T. Leveraging pre-commit hooks for context-sensitive checklists: a case study. In *Fachtagung des GI-Fachbereichs Softwaretechnik, Software Engineering (SE 2015), Dresden, Germany* (2015), pp. 219–222. [28]

Invited Talks

- ISEC 2018, Hyderabad, India: "Factors Influencing Code Review Processes in Industry – Extended Version"
- 56th CREST Open Workshop, 2017, UCL, London, UK: "Culture is Key: Results of a Survey on Factors Influencing Code Review Adoption"

Funding and Awards

- ACM SIGSOFT Distinguished Paper Award at FSE 2016 for "Factors Influencing Code Review Processes in Industry" [39]
- Best Paper Award at ICSSP 2016 for "Comparing Pre Commit Reviews and Post Commit Reviews Using Process Simulation" [34]
- Scholarship 'Studienstiftung des deutschen Volkes', 2006 – 2009
- 22. Bundeswettbewerb Informatik, 2003/2004

---

[1]Extended version of [34]

- Reached final round
- Special price 'originellste Einzelidee'
- Special price 'beste Gruppenleistung'