Factors Influencing Code Review Processes in Industry

Tobias Baum, Olga Liskin, Kai Niklas and Kurt Schneider
FG Software Engineering
Leibniz Universität Hannover
Hannover, Germany
{forename.surname}@inf.uni-hannover.de

ABSTRACT

Code review is known to be an efficient quality assurance technique. Many software companies today use it, usually with a process similar to the patch review process in open source software development. However, there is still a large fraction of companies performing almost no code reviews at all. And the companies that do code reviews have a lot of variation in the details of their processes. For researchers trying to improve the use of code reviews in industry, it is important to know the reasons for these process variations. We have performed a grounded theory study to clarify process variations and their rationales. The study is based on interviews with software development professionals from 19 companies. These interviews provided insights into the reasons and influencing factors behind the adoption or non-adoption of code reviews as a whole as well as for different process variations. We have condensed these findings into seven hypotheses and a classification of the influencing factors. Our results show the importance of cultural and social issues for review adoption. They trace many process variations to differences in development context and in desired review effects.

Keywords

Code inspections and walkthroughs, Software engineering process, Software process, Code reviews, Empirical software engineering

1. INTRODUCTION

Starting with Fagan's influential study [12], there is a lot of evidence that Inspections and other types of code review are efficient software quality assurance techniques. Nevertheless, they are not adopted throughout the whole industry. A survey by Ciolkowski, Laitenberger and Biffl from 2002 [6] found a share of 28% of the participants doing code reviews. We believe that this figure has increased in the meantime, but the basic question remains: Why is not everybody doing code reviews?

FSE 2016 Seattle, USA

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

In many companies, the review process is converging towards a lightweight process that is based on the regular review of changes [29] and that is similar to the processes used by many open source projects [30][1]. Looking beyond the convergence in the general structure of the review process, we found a lot of variation in the details, a fact also observed in earlier studies [17]. Which part of these variations is accidental, and which factors influence the choices taken by industrial practitioners?

Our long-term research objective is to improve the use of code reviews in industry. A thorough understanding of the reasons for the adoption or non-adoption of code reviews and for the choices taken between several possible review variants will help to guide this research and to predict the consequences and limitations of our work. Therefore, we performed a grounded theory study based on interviews with 24 software engineering professionals from 19 companies. The broad *research questions* treated in this article are:

- RQ1: Why are code reviews used (or not used) by industrial software development teams?
- RQ2: Why are code reviews used the way they are?

We describe our main contributions in section 3.3: The structuring of the influencing factors and the formulation of grounded hypotheses regarding the research questions. Before that, we describe the chosen methodology (section 2) and some necessary background information from the study (sections 3.1 and 3.2). We also compare our results to other researchers' work (section 4), and discuss the limitations of our study (section 5) as well as the implications of our results (section 6).

2. METHODOLOGY

Our goal is to build theory (in the sense of connected hypotheses) regarding the factors influencing review process decisions. The "Grounded Theory" methodology [1, 10, 15] suits these goals well: It uses an iterative approach to build a theory that is "grounded" in data. We have chosen interviews with industrial practitioners as our primary source of data because they are well-suited to elicit motivations, opinions and detailed descriptions. To clarify our interpretation

[1]This process is often referred to as "contemporary code review" or "modern code review". We prefer the terms "continuous differential code review" [4] or "change-based code review", because they are less judgmental and more self-explanatory. One of the earliest publications describing such a differential code review process is from Baker [3].

Table 1: Demographics and Review Use of Companies

ID	type	employees (IT, approx.)	country	dev. process	spatial	regular review use
A	in-house IT, travel	450	DE	agile	co-located	no
B	standard software, dev. tools	400	CZ	ad hoc	distributed	yes
C	standard software, government	200	DE	classic / ad hoc	co-located	no
D	standard software, CAD	100	DE	ad hoc	co-located	yes
E	standard software, output mgmt.	70	DE	agile	co-located	yes
F	standard software, agriculture	130	DE	agile	co-located	yes
G	standard software, retail	50	DE	agile	co-located	yes
H	contractor, automotive	70	DE	agile	co-located	yes
I	SaaS, dev. tools	5	US	ad hoc	distributed	no
J	in-house IT, finance	1100	DE	ad hoc	co-located	no
K	in-house IT, finance	200	DE	classic	co-located	no
L	in-house IT, finance	400	DE	classic	co-located	yes
M	in-house IT, government	200	DE	classic	co-located	no
N	in-house IT, marketing	50	DE	agile	co-located	no
O	–	–	DE	–	co-located	yes
P	in-house IT, finance	–	DE	–	distributed	yes
Q	in-house IT, retail	120	DE	classic / agile	co-located	yes
R	in-house IT, marketing	50	DE	agile	co-located	no
S	in-house IT, automotive	4000	DE	agile	co-located	yes

of "Grounded Theory", we will describe our method in more detail in the following.

We performed "theoretical sampling" to select the interviewees: We used our emerging theory to choose participants that could extend or challenge this theory, for example because they came from a so far little investigated context. We gained access to them 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, we performed 22 interviews with 24 participants. They described 22 different cases of code review use in 19 companies. Four interview requests did not result in an interview. Detailed information on participants' demographics and contextual information can be seen in tables 1 and 2[2]. Our sample has a focus on small and medium standard software development companies and in-house IT departments from Germany, but we included contrasting cases for all main factors. As some examples, company S is much larger and company I much smaller than the other cases. Companies B and I do the main development outside of Germany. The team interviewed from company J 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, as the development teams were responsible for code reviews in the sampled cases. We sampled several interviewees from company E. This was done in part to test to what extent descriptions of the same underlying process differ between participants, but also to gain insight into the work of single reviewers. The second topic is out of the scope of the current article. 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 were semi-structured, using open-ended questions. The corresponding interview guide was initially created based on the research questions and checked by another researcher who has experience with interview studies and Grounded Theory. It was later continually adjusted according to earlier interview experiences and the emerging theory[3]. The interviews lasted 46 minutes on average, ranging from 24 minutes to 78 minutes. We preferred face-to-face interviews and used Skype or telephone for 5 interviewees where a face-to-face interview was not possible. Three participants (IDs 19, 20 and 21 in table 2) were interrogated in a group interview[4], all other interviews were conducted with single persons. All interviews were recorded and later transcribed. Most interviews were conducted by the first author. Some of the interviewees were colleagues of the first author. To reduce the resulting bias as a potential threat to credibility, these particular interviews were performed by the second author. We ensured anonymity to all participants. For this reason, the interview transcripts are available only upon request to the first author.

After performing and transcribing the first four interviews, we started data analysis: We used open coding to identify common themes in the data and analyzed the resulting codes for dimensions in which they vary as well as similarities. Coding was done paper-based at first and later using Atlas.TI [14]. Coding was done incrementally and iteratively,

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

[3]The first and last used versions of the interview guide are available online:
http://tobias-baum.de/rp/interviewGuideFirst.pdf,
http://tobias-baum.de/rp/interviewGuideLast.pdf

[4]Sadly, the gate-keeper which had provided access to these interviewees left the company during our study. We could not reach them by other means. Therefore some entries in table 1 and 2 could not be determined and had to be left blank.

Table 2: Demographics of Interviewees

Comp.	ID	role	industrial SD experience (years)
A	1	software developer	30
B	2	software developer	15
C	3	software developer	15
D	4	software developer	17
E	5	team/project lead	10
	6	software developer	25
	7	software developer	10
	8	software developer	7
	9	software developer	7
	10	software developer	6
F	11	team/project lead	12
G	12	team/project lead	10
H	13	team/project lead	15
I	14	team/project lead	14
J	15	software developer	16
K	16	software developer	3
L	17	software developer	6
M	18	req. engineer (consultant)	20
N	19	software dev. (consultant)	20
O	20	team/project lead (cons.)	–
P	21	team/project lead (cons.)	–
Q	22	software developer	3
R	23	software developer	14
S	24	team/project lead (cons.)	18

including new interviews as they were taken and revisiting most interviews several times. In this constant comparison process [10], we compared and related citations and codes to each other. Furthermore, we contrasted whole cases of review usage to carve out their differences and similarities. The basic open coding was done by the first and third author. The results were compared and discussed afterwards to check for possible bias or different viewpoints. During the whole process, memos were written to capture emerging ideas. Memoing was intensified towards the end of the study, capturing descriptions of the codes and the corresponding hypotheses. All authors engaged in discussing the memos as well as checking for consistency with the raw data. Finally, 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 our confidence that our results are a suitable description of the participants' reality.

Following Grounded Theory practices, we performed the main literature review after collecting our study results. In section 4, we compare our theory to results from similar studies as well as to more general sociological theories.

3. RESULTS

In section 3.1, we give an overview over the process variants we observed in our interviews. This is not an exhaustive overview, as it merely serves to provide the background for the hypotheses we will present in section 3.3. Many of these variations have also been observed in previous studies on contemporary code review practices [30]. In section 3.2 we describe the effects that our interviewees attributed to code reviews, also as a prerequisite for section 3.3. A similar set of expected effects was observed in [2].

3.1 Overview of the Observed Processes

Looking at how reviews are embedded into the development process, we could observe two review variants: *Irregular, non-systematic code review* and *regular differential code review*.

Irregular, non-systematic review This kind of review is usually triggered when an individual (mostly the author) feels the need for a review. None of the studied teams stated to do no review at all, but in many cases irregular review goes hand in hand with low review usage. The specifics for this type of review vary widely based on the triggerers' intention. Most often their intention is a concrete need for relief or help with a problem by a second person, mostly an expert. The process is then streamlined to solve this specific problem. Having a classical definition of "review" in mind, one can question if this really is a proper kind of review at all.

Regular, differential code review Unlike irregular review, this kind of review is codified in the development process of the team or organization: Every time a "unit of work" is seen as "done", all changes that were performed in the course of its implementation are considered a review candidate. According to rules specified in the process, it is assessed which parts of the candidate changes need to be reviewed. Our term "unit of work" is similar to the "patch set" identified in other studies. We did not use "patch set" because it seemed more narrowly focused on specific technologies.

Of the studied companies, eleven have a regular code review process, whereas the remaining eight only do irregular code reviews. The last column of table 1 shows this in more detail.

For irregular reviews, there is no codified review process for a team. Therefore the description of process variants given below and the influencing factors for the choice of a process variant only contain data from cases with regular reviews. On the other hand, our hypotheses on the use or non-use of reviews include data from all sampled cases.

The studied cases differ in the type of "unit of work" chosen as a trigger for reviews: Some teams use larger units, like user stories or requirements, others use smaller units like development tasks or even single commits. The triggering of reviews is often supported by tools. This can be a separate state in a bug tracker's ticket workflow or the usage of specialized tools that enforce this process, like pull requests on GitHub[5] or Gerrit[6].

Another difference between the studied cases is whether the code is reviewed before or after it is published to the development trunk. The first type is commonly called "pre commit review" or "Review Then Commit", the second type

[5]https://github.com

[6]https://www.gerritcodereview.com

"post commit review" or "Commit Then Review". "Pull requests" are a special form of pre commit reviews that has recently gained popularity.

All studied teams that perform regular reviews to detect defects try to perform all reviews before the changes are made available to customers. They use different means to reach this goal:

- Increasing the priority of reviews and ensuring swift completion of reviews by organizational means.
- Demanding the use of pre commit reviews, either generally or for a certain time before releases.
- Using a technical separation between development branch/stream and release branch/stream.

The studied cases also differ in the choice of reviewers. In some teams, everybody can or even should review everything, in other teams only a subset of the developers acts as reviewer. The usual number of reviewers (excluding the author) in the studied cases is between one and two. In some of the cases the number is fixed, in others it depends on factors that are similar to the ones determining if a review is performed at all: A higher estimated complexity of the change (identified by the author or already during task planning) or more severe estimated consequences of defects: *"When it's database migration code [. . .], five to six selected reviewers have to give their OK to it" (I. 20)*[7].

We encountered significant variations regarding the use of interaction in reviews. Permanent face-to-face interaction between the reviewers or between reviewers and author forms one end of this spectrum. The other extreme is purely asynchronous, electronic, on-demand communication on the issues found.

A final variation we will touch upon in this article concerns "fixing on the fly": In some teams, it is usual that reviewers fix minor, low risk issues on their own while reviewing, while other teams allow no code changes during review.

3.2 Review Effects

Our interviewees expect reviews to have certain effects. We found that reaching desired and also avoiding undesired effects was a main influence in choosing process variants. Therefore this section provides a short overview of the effects 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"). We will not describe those single developer effects, as they are of minor importance for the rest of our discussion. An overview of the team level effects is shown in figure 1.

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).

[7]The numbers given at the citations from the interviews refer to the interviewee ID from table 2. We use English translations for all citations, although the interviews were done in German. The original statements are available upon request.

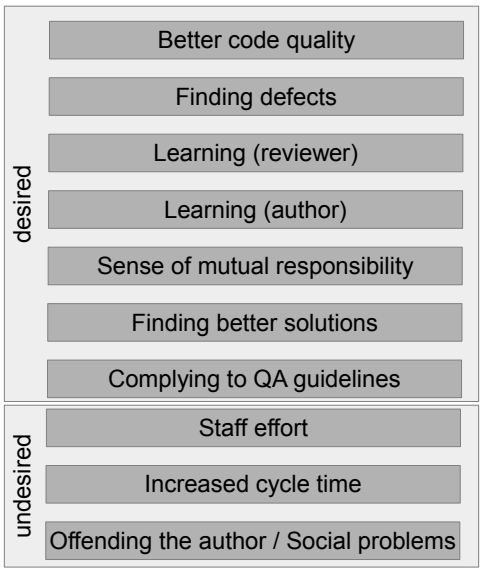

Figure 1: Overview of found desired and undesired review effects.

Finding defects (desired) Reviews are expected to find defects (regarding external quality aspects, especially correctness). This is seen as especially 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)* Additionally, they learn something about the reviewers' values and their 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.

Offending the author (undesired) The author can feel offended (or discouraged) when his or her code is reviewed or when issues are found. While 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)*

3.3 Hypothesis generation: What determines the Review Process?

This section contains our main results. As we use a methodology that is theory generating, they take the form of grounded hypotheses. The subsections 3.3.1 and 3.3.2 describe what triggers and inhibits review use (RQ1). The remaining subsections 3.3.3 to 3.3.6 then discuss the influences leading to different review process variations (RQ2). To provide reference from statements to interviews we use the interviewee ID from table 2 as a subscript.

3.3.1 Triggers of review introduction

Introduction of reviews was done mostly as a reaction to a problematic incident: There had been quality issues regarding correctness or maintainability [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 [11,13] or some changes were perceived as risky or insecure [4,21]. Another trigger is demand for code reviews by an external stakeholder [19], or positive experiences with reviews in the past [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 [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 [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 [2,13] or a professional movement like "clean code development" [10]), the product/project context [2,4,8,18,20] (e.g. "Which quality level is needed for the customer?"), the team members' personalities [1,11,13,18] and the team's culture [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 [7,11,24] (e.g. agile "retrospectives").

> HYPOTHESIS 1. *Code review processes are mainly introduced or changed when a problem, i.e. a gap between some goal and reality, is perceived.*

3.3.2 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 [9,10,15].

Another category of inhibitors are problems that are generally associated with change: There is resistance to change among the people concerned [16]. Lack of knowledge and corresponding insecurity [16,18] belong to this category, too. Furthermore, performing a process change consumes effort, leading to conflicts with other projects and tasks [12,18]. The effort for a process change varies widely and can be substantial in bureaucratic organizations [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 [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 [4,15,22,23].

> HYPOTHESIS 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.*

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 [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 our observation that while social problems are the top reason to refrain from starting reviews, a negative assessment of costs versus benefits dominates when reviews are stopped again [4,19,20].

HYPOTHESIS 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 as *"fading away"* [1,3,7,13,19] and was seldom an explicitly stated directive. This 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 [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.

HYPOTHESIS 4. *Code review is most likely to remain in use if it is embedded into the process (and its supporting tools) so that it does not require a conscious decision to do a review.*

3.3.3 Factors shaping the process

When a change in (or introduction of) the review process is triggered, a small number of possible solutions is examined [5,7,17,24] (see also section 3.3.6). Each possibility has to satisfy mainly two criteria: It has to fit into the context of the team [4,5,9,12,17,24], and it has to be believed to provide the expected desired effects [3,6,8,9,10,14,19,21,22,23] while staying in the acceptable range of undesired effects [4,6,7,9,10,14,18,19,21,22]. These general relationships are visualized in figure 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 [10,12,13]. We did not include influences between the contextual factors into the diagram.

We divided the factor forming the context into five categories: "Culture", "development team", "product", "development process" and "tool context". In the following we will describe the factors that we extracted from the interviews. We will also give examples for their influence on the review process.

The category "culture" subsumes customs, values and beliefs of the team or company. We found the following interrelated factors belonging 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 [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 measure of knowledge distribution [11]. Many interviewees believe that face-to-face communication is better suited for knowledge distribution than written forms [4,7,8,9,11,21].

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 [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 [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 [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 [2,4,5,9,11,14,22]. Therefore, some teams choose to restrict the reviewer population to experienced team members [12,22,24]. Less experienced team members often introduce more defects and benefit more from knowledge transfer through reviews [14,19,21,24].

Spatial distribution Some teams work co-located, others are distributed. In distributed teams, face-to-face interaction in reviews is nearly impossible [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 [24].

Contractual or legislative obligations When code reviews are mandated by a contract or law, the review process is designed to satisfy these requirements [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 [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 decide between tasks and stories as the unit of work to review when both are available in the development process.

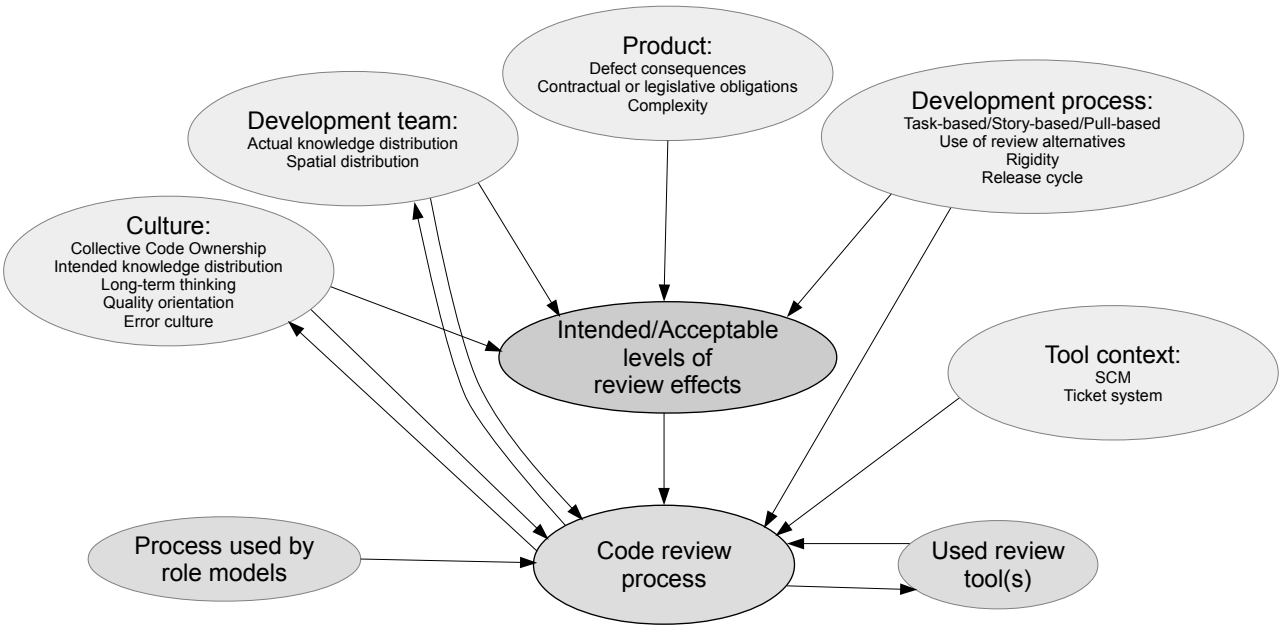

Figure 2: Main factors shaping the review process. Arrows mean "influences".

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 [6,12,14,20,23,24]. Many teams use static code analysis on a continuous integration server to detect maintainability issues [2,3,6,8,11,23,24]. To find better solutions, many teams discuss requirements and design alternatives before code review [5,6,7,23]. Pair programming is another alternative to code reviews, often seen to provide similar benefits [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 [2].

Release cycle When releases are very frequent, the acceptable increase in cycle time through reviews is lower [23]. Frequent or even "continuous" releases also demand techniques to keep unreviewed changes from being delivered to the customer. This often means pre-commit reviews/pull requests [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 [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 [12].

3.3.4 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 2). 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 heavily 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 5. *The intended and acceptable levels of review effects are a mediator in determining the code review process.*

For reasons of space and conciseness, we did not describe every relationship between a review factor and a process variant in section 3.3.3. More detailed results are available online[8].

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 we described in the last paragraphs 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 [13,22,24].

3.3.5 Sources of information

Until here, we mainly described how the choice among the possibilities is performed, but it remained open how the

[8]http://tobias-baum.de/rp/detailedTable.pdf

possibilities to examine are determined: We asked the interviewees 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 3,8,12,16,17,18. The information they had was gained mostly rather by chance from these sources:

- colleagues or other teams in the same company 2,13
- open source projects 9
- blogs and web pages 1,10,21
- university education 14
- practitioners' journals 1,15,18,24
- practitioners' conferences 24
- books on software engineering best practices 15,24

Of these, own experiences in open source projects and experiences from colleagues were the most influential 2.

3.3.6 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 12. When choosing a review process, the number of examined possibilities was often quite low 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 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" 10,12,13. This observation is similar to the one that led to *hypothesis 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" we observed several times B,E,G,H,S: 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. *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.*

4. RELATED WORK

Our results confirm some previous studies and challenge or extend others. In the following, we discuss our results in comparison to some of the related work as well as to some more general theories.

4.1 Theories on the Adoption and Shaping of Reviews

Harjumaa et al. [17] studied characteristics of and motivators and demotivators for peer reviews. The study is based on 12 development divisions from 10 small Finnish software companies. Interviews were performed in a rather structured fashion, based on a questionnaire developed using information from the literature, and analyzed mostly quantitatively. The authors report large differences in the review processes used, but don't analyze these differences in detail. They also 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.

The code review process at Microsoft has been studied by Bacchelli and Bird [2][9]. They used a mixed methods approach, based on interviews, review comments and surveys to shed light on the motivations for and challenges in performing tool-based differential code-reviews. Their results indicate that finding functional defects is only one among many motivations for performing code reviews, albeit an important one, which supports our result that a combination of review effects is responsible for the review process chosen. A similar result was found by Mantyla and Lassenius [22] when they studied the types of defects found in code review. In another recent case study [35], Spohrer et al. put a focus on knowledge creation through code reviews and pair programming.

In a partially qualitative study on open source review practices, Rigby and Storey [31] analyzed the mechanisms behind review decisions. They found that experience and interest of the reviewer are a major factor in deciding whether to review a patch. Many of their findings are mainly relevant to open source development processes.

Many other current theories on the use or non-use of code reviews are largely anecdotal. In [13], Fagan summarizes some of his experiences. He names a general aversion to change and a fear of too much effort (*not* too much effort itself) as reasons hindering the adoption of reviews. When Jalote and Haragopal [19] make the "not applicable here" syndrome responsible for the non-use of reviews, this is a special case of a cultural issue and therefore of *hypothesis 2*. Social problems with inspections were also described by Iisakka and Tervonen [18], based on their consulting experiences.

4.2 General behavioral theories

In his book "Diffusion of Innovations" [32], 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. This supports our finding that review processes of familiar teams have a large influence (*hypothesis 6*). 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. Given these categories, the "intended review effects" we

[9]later extended and summarized by Rigby et al. in [28]

identified as major influencing factors 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 we found in our 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 [23]. 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. Our 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 5*).

4.3 Theories on the Adoption and Shaping of Software Process Improvements

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 our results to general studies on this topic.

In Coleman's study of software process formation and evolution in practice [8, 9], he observed that process change is triggered by "business events", in most cases problems. This is similar to our *hypothesis 1*. After a change, "process erosion" occurs until a "minimum process" is reached. In the cases we studied, major erosion occurred only when code reviews had not been institutionalized (*hypothesis 4*). Like us, 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 we 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. Similar to our *hypothesis 6*, he notes that the initial process used in start-up companies is largely based on a model process stemming from the manager's experiences.

Clarke and O'Connor combined several studies to develop a reference framework of factors affecting the software development process [7]. Compared to our classifications, 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 [33], by Mustonen-Ollila and Lyytinen using diffusion of innovation theory as a guiding framework to quantify some aspects of the adoption of IS innovations [24] and by Orlikowski on the adoption of CASE tools [26].

4.4 Other related work

Our findings show that the review process is tailored according to the pursued goals (*hypothesis 5*). Tailoring of review processes has also been proposed in the research literature, for example in the TAQtIC approach [11] with a focus on classical Inspection. And Green et al. [16] noted that "[the pursued] value can change everything" with regard to tailoring software processes.

Some other sources also support *hypothesis 4*: Komssi et al. describe experiences at F-Secure [20] where document inspections were not integrated into the process and were abandoned as soon as the champions lost their interest. And Land and Higgs [21] describe the institutionalization of development practices in a case study of an Australian software company.

Porter et al. [27] observed that only a small share of the variance in review performance can be explained by process structure, and Sauer et al. [34] arrive at a similar result by theoretical considerations. This indicates that the choice to give little attention to the review processes' details (*hypothesis 6*) is likely a sensible one.

5. LIMITATIONS

According to Charmaz [5], the quality of a grounded theory can be assessed based on four categories: Credibility, originality, resonance and usefulness. Regarding originality, the discussion in section 4 showed that some of our hypotheses and their combination and application to code reviews are new, but also that some of the hypotheses can be seen as specializations of more abstract theories. Regarding usefulness, we will describe some implications of our results to researchers and practitioners in section 6. In the current section, we will mainly discuss limitations and threats to credibility as well as the measures taken to mitigate these. We used the catalog provided by Onwuegbuzie and Leech [25] as an additional source for identifying threats to credibility.

A characteristic that our study shares with all Grounded Theory studies is that it relies on the human researcher as instrument for data collection and analysis and is therefore prone to researcher bias. We tried to mitigate this threat by following Grounded Theory best practices, a reflexive approach to our research [5] and additional measures described below.

Some threats to validity result from our choice of interviews as main method of data collection. There is a risk that the interviewees left out or changed some aspects of their processes when describing them. The interviews did not touch upon sensitive personal data, but nevertheless we tried to reduce this risk by ensuring anonymity to all participants. The validity of the interview data could also be threatened by asking for descriptions and rationales after the fact. The interviewees had to recall triggers for process changes, and there is a risk that they are rationalizing their decisions in retrospect. We checked for consistency using triangulation of several descriptions of the same process at company E and authors' observations at companies E and J. This led us to conclude that this risk is low enough. Nevertheless, the reader should keep in mind that there is only one data point for most of the companies and that a larger study using several data points from each company as well as longitudinal observation could provide more reliable results.

The choice of interviewer can also result in bias. To mitigate this risk, we used an interview guide and asked another experienced researcher to check this guide before using it. As seven of the interviewees were colleagues of two of the authors, we made sure that these interviews were conducted by another one of the authors so that passive researcher bias was reduced.

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, we explicitly asked for points missed in the discussion at the end of each interview. By recording and transcribing all interviews, we ensured that no information was lost unintentionally and that we could come back to the data for coding and checking several times. We also tried to account for the enlarged share of data from company E that resulted from our choice to use several data points from this company for triangulation.

To mitigate researcher bias during data analysis and interpretation, coding was performed by several authors and the results were discussed by all authors. As another important mitigation for observational as well as researcher bias, we performed "member checking": We provided our results to the study participants and asked for feedback, which was then incorporated into the study. This member checking also assured the "resonance" of our results.

As Grounded Theory studies use theoretical sampling and rather small sample sizes, it is hard to assess their generalizability. We used a broad range of different interviewees from a heterogeneous sample of companies, and our sample size is quite large compared to most qualitative studies. The most significant biases we are aware of is that most of the companies and all of the interviewees are German, that we did not include teams producing highly safety critical software and that we did not include interviewees from upper management.

6. DISCUSSION

Our results have implications for researchers developing tools for code reviews: They show that a "one size fits all" tool is not likely to succeed, but they also show the way in which tools influence review processes. Our results also indicate that further improvements in review efficiency or effectiveness will not lead to large increases in review adoption, as long as there is no change in cultural and social factors.

Regarding future work, our results should be extended and tested with further observations. We are currently preparing to complement our results using quantitative methods. We studied the adoption of reviews much more deeply than their abandoning. More data could be collected to shed more light into how this happens. Additionally, we plan to further analyze the interview data we already gathered to study individual reviewers' problems and habits.

One of the results of our study is that the probability of an industrial practitioner reading this text is quite low. Nevertheless, we want to discuss some of the implications of our results to practitioners as well: Our results show that the process a team starts with tends to stick. Therefore it pays off to make oneself aware of the goals one is trying to reach with the aid of reviews, and to choose corresponding process variants right from the start. Finally, every team not doing reviews at the moment due to fear of social problems should take *hypothesis 3* into account: If reviews seem beneficial otherwise, choose a process that minimizes negative social effects and just try it for some time. And ensure that reviewing is embedded into the development process, so that it will stay in use in the long term.

7. CONCLUSION

We performed a study to examine code review use in commercial software development teams. In this article we present the resulting theory on why code reviews are used (or not used) in the way they are. Our study is based on Grounded Theory methodology. We interviewed 24 participants from 19 companies and performed an in depth analysis of these interviews.

We condensed our findings on the adoption of reviews into the following hypotheses:

- *Hypothesis 1*: Code review processes are mainly introduced or changed when a problem, i.e. a gap between some goal and reality, is perceived.
- *Hypothesis 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 3*: The importance of negative social effects decreases with time when reviews are in regular use.
- *Hypothesis 4*: Code review is most likely to remain in use if it is embedded into the process (and its supporting tools) so that it does not require a conscious decision to do a review.

The interviewees mostly relied on their own or colleagues' experiences and considerations to gain knowledge on review processes. Web pages and practitioners' journals and conferences were also used, while scientific journals or conferences were not used at all.

We identified several contextual factors that influence the review process and grouped them into five categories: "Culture", "development team", "product", "development process" and "tool context". They influence the review process directly, mainly by making certain process variants infeasible or unattractive. But a major influence is also exerted indirectly, mediated through the intended combination of review effects. This does not mean that every process choice is made thoughtfully: Many minor process aspects just stay the way they were first tried. We formulated the following hypotheses for this subject area:

- *Hypothesis 5*: The intended and acceptable levels of review effects are a mediator in determining the code review process. A classification of the influencing factors is depicted in figure 2.
- *Hypothesis 6*: 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.

8. ACKNOWLEDGMENTS

The authors would like to thank all participants for the time and effort they donated for the interviews and for member checking. Further thanks go to Raphael Pham for his methodological advice and valuable feedback and to Matthias Becker for his remarks on the article's draft.

9. REFERENCES

[1] 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.

[2] 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.

[3] R. A. Baker Jr. Code reviews enhance software quality. In *Proceedings of the 19th International Conference on Software Engineering*, pages 570–571. ACM, 1997.

[4] M. Bernhart and T. Grechenig. On the understanding of programs with continuous code reviews. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pages 192–198. IEEE, 2013.

[5] K. Charmaz. *Constructing Grounded Theory*. Introducing Qualitative Methods series. SAGE Publications, 2014.

[6] M. Ciolkowski, O. Laitenberger, and S. Biffl. Software reviews: The state of the practice. *IEEE software*, 20(6):46–51, 2003.

[7] 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.

[8] 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.

[9] 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.

[10] J. Corbin and A. Strauss. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE Publications, 3e edition, 2007.

[11] C. Denger and F. Shull. A practical approach for quality-driven inspections. *Software, IEEE*, 24(2):79–86, 2007.

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

[13] M. Fagan. A history of software inspections. In *Software pioneers*, pages 562–573. Springer, 2002.

[14] S. Friese. *Qualitative Data Analysis with ATLAS.ti*. SAGE Publications, 2012.

[15] B. Glaser and A. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine, 1967.

[16] 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.

[17] 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.

[18] J. Iisakka and I. Tervonen. The darker side of inspection. In *Proceedings of Workshop on Inspection in Software Engineering (WISE 2001), Paris*, 2001.

[19] P. Jalote and M. Haragopal. Overcoming the nah syndrome for inspection deployment. In *Proceedings of the 20th International Conference on Software Engineering*, pages 371–378. IEEE Computer Society, 1998.

[20] 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.

[21] 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.

[22] 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.

[23] J. G. March and H. A. Simon. *Organizations*. John Wiley & Sons, Inc., 1958.

[24] 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.

[25] A. J. Onwuegbuzie and N. L. Leech. Validity and qualitative research: An oxymoron? *Quality & Quantity*, 41(2):233–249, 2007.

[26] W. J. Orlikowski. Case tools as organizational change: Investigating incremental and radical changes in systems development. *MIS quarterly*, pages 309–340, 1993.

[27] 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.

[28] P. C. Rigby, A. Bacchelli, G. Gousios, and M. Mukadam. A mixed methods approach to mining code review data: Examples and a study of multi-commit reviews and pull requests. In C. Bird, T. Menzies, and T. Zimmermann, editors, *The Art and Science of Analyzing Software Data*. Morgan Kaufmann, 2015. in press.

[29] 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. ACM, 2013.

[30] 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.

[31] 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.

[32] E. M. Rogers. *Diffusion of Innovations*. Free Press,

5th edition, 2003.

[33] 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*, pages 1–22, 2015.

[34] 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.

[35] K. Spohrer, T. Kude, C. T. Schmidt, and A. Heinzl. Knowledge creation in information systems development teams: The role of pair programming and peer code review. In *ECIS*, page 213, 2013.