Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Associating Working Memory Capacity and Code Change Ordering with Code Review Performance

Tobias Baum · Kurt Schneider · Alberto Bacchelli

Received: date / Accepted: date

Abstract Change-based code review is a software quality assurance technique that is widely used in practice. Therefore, better understanding what influences performance in code review and finding ways to improve it can have a large impact. In this study, we examine the association of working memory capacity and cognitive load to code review performance and we test the predictions of a recent theory regarding improved code review efficiency with certain code change part orders. We perform a confirmatory experiment with 50 participants, mostly professional software developers. The participants performed code reviews on one small and two larger code changes from an open source software system to which we had seeded additional defects. We measured their efficiency and effectiveness in defect detection, their working memory capacity, and several potential confounding factors. We find that there is a moderate association between working memory capacity and the effectiveness of finding delocalized defects, influenced by other factors, whereas the association with other defect types is almost non-existing. We also confirm that the effectiveness of reviews is significantly larger for small code changes. We observe a tendency that the order of presenting the code changes influences the efficiency of code review, but cannot conclude this reliably.

Keywords Change-based Code Review · Working Memory · Individual Differences · Code Ordering · Cognitive Support · Cognitive Load

Tobias Baum / Kurt Schneider
Fachgebiet Software Engineering
Leibniz University Hannover
Germany
E-mail: {firstname.lastname}@inf.uni-hannover.de

Alberto Bacchelli
ZEST
University of Zurich
Switzerland
E-mail: bacchelli@ifi.uzh.ch

1 Introduction

Code Review, *i.e.*, the reading and checking of source code by developers other than the code's author (Fagan, 1976), is a widespread software quality assurance technique (Baum et al, 2017a). In recent years, code review in industry has moved from traditional Fagan Inspection (Fagan, 1976) to 'change-based code review' (Baum et al, 2016a; Rigby and Bird, 2013; Bernhart and Grechenig, 2013), in which teams determine the review scope based on code changes, such as commits, patches, or pull requests (Gousios et al, 2014).[1]

Code review performance is traditionally measured as the share of defects found (*effectiveness*) and defects found per unit of time (*efficiency*) (Biffl, 2000). Past research (*e.g.*, Porter et al (1998); Bacchelli and Bird (2013)) has provided evidence that code review performance is determined to a large degree by human factors, thus current research efforts focus on investigating how automated tooling can help humans performing reviews with higher efficiency and effectiveness (Baum and Schneider, 2016; Thongtanunam et al, 2015b; Balachandran, 2013; Tao and Kim, 2015; Barnett et al, 2015; Kalyan et al, 2016).

A large portion of the research efforts on automated tools to help code reviewing is explicitly or implicitly based on the assumption that reducing the *mental load* of reviewers improves their code review performance (Baum et al, 2017b). Cognitive psychology uses the term mental load to denote the aspects of a task (and environment) that influence its *cognitive load*, *i.e.*, the workload on the human cognitive system (Paas and Van Merriënboer, 1994). Cognitive load is also influenced by factors depending on the human, like the working memory capacity and the invested mental effort. Cognitive load, in turn, can influence task performance, especially in situations of overload. In other words, since reviewers need to understand the code change under review and relate its parts to other code portions and to background information, automated tooling for code review is based on the assumption that the more cognitive resources are available for these tasks, the better performance should be. Mental/cognitive load is not the only framework used to guide research on improved review tooling. Examples of competing frameworks are "activating the reviewer" (*e.g.*, Denger et al (2004)) or "discipline in systematic reading" (*e.g.*, Basili et al (1996)). Which of these frameworks, and in which situations, is most adequate and to what extent is still an open question.

In the current study we investigate a selection of practically relevant aspects of the cognitive load framework: If the cognitive load of relating code parts to each other is associated with code review performance, reviewers with a higher working memory capacity should perform better under high load, with implications for reviewer selection and teaching. Mental load will increase with the size and complexity of the code change, and therefore review performance should be lower for more complex code changes. Furthermore, our recent re-

[1] For consistency, we will stick to the term 'code change' (Baum et al, 2017b) throughout this article. We could also use 'patch', since every code change in the study corresponds to a single patch.

search (Baum et al, 2017b) proposes that automatic improvement of the order in which the parts of the code change (*e.g.*, changed methods or changed files) are presented to the reviewer can reduce mental load and therefore increase review performance.

To investigate these hypotheses, we performed an experiment in which 50 software developers (46 being professionals) spent a median time of 84 minutes each, performing one small and two large change-based reviews. We measured time spent, found defects, as well as a subjective assessment of relative mental load; for 45 of the participants we also measured the working memory capacity.

We find that working memory capacity is associated with aspects of the code review performance, namely the effectiveness of finding *delocalized defects*[2]; while there is only a very weak association with other defect types. We obtain confirmatory evidence that larger, more complex changes are associated with lower defect detection effectiveness, thereby triangulating results from earlier studies. For the effect of change part ordering, our findings are less conclusive: There is insufficient evidence to allow robust conclusions that change part ordering has an effect on code review performance. Our findings show that the general tendency and the qualitative data is compatible with the predictions in our previous work (Baum et al, 2017b), but we do not reach statistical significance. Not having attained the sample size indicated by power analysis could have had an effect on this result.

We regard the results regarding working memory capacity as the most important of our findings: Firstly, they help to understand cognitive processes during code review. Secondly, when interpreted in a negative sense ("there is an association, but it is not extremely strong and only for certain defects"), they indicate that reviewers with average cognitive abilities can still deliver valuable review results and they support the use of heterogenous review teams in practice.

Summing up, this study makes the following contributions:

- The first study on the association between working memory capacity and code review effectiveness, which provides evidence that working memory capacity is only associated with effectiveness for certain kinds of defects.
- Confirmation from a controlled experiment that review effectiveness is higher for smaller code changes.
- Empirical data on the influence of change part ordering on mental load and review performance. We do not reach statistical significance in this regard, but future meta-analysis can build upon this data.
- A dataset (Baum et al, 2018) of the review sessions belonging to the experiment as a foundation for further research, containing, for example, detailed review results and navigation patterns.
- Experiences with a research design for performing an experiment online in a browser-based setting (which we used to ease access to professional

[2] defects for which knowledge of several parts of the code needs to be combined to find them

developers and to control the ensuing threats to validity) and its software implementation (Baum et al, 2018).

We hope that our results will be a stepping stone towards a better understanding of the cognitive processes during code review and that they can stimulate further research on improved review tooling, code reviewer selection, and code review education.

2 Background and Related Work

In the following, we describe key terms from previous work that we are building upon, like working memory, cognitive and mental load, delocalized defects and ordering of code for review. We also survey existing findings in related areas. For easier reference, Table 1 at the end of this section shows a summary of important terminology we introduce.

2.1 Cognitive Load and Working Memory

Cognitive load is "a multidimensional construct that represents the load that performing a particular task imposes on the cognitive system" (Paas and Van Merriënboer, 1994). 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.

Cognitive load is used in various theories that explain performance on cognitive tasks, *e.g.*, cognitive load theory for learning (Sweller, 1988), which predicts that better learning success is achieved by avoiding extraneous cognitive load.

The capacity of human *working memory* (Wilhelm et al, 2013) is a major determinant of 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 employed 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 (Cowan, 2010). To overcome this limitation, items can be combined by 'chunking' (Simon, 1974; Miller, 1956) 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' (Daneman and Carpenter, 1980), 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 (Daneman and Carpenter, 1980; Daneman and Merikle, 1996) and hypertext (DeStefano and LeFevre, 2007). In the context of software engineering, Bergersen and Gustafsson (2011) 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 (2016) found reduced cognitive load during code understanding for more experienced participants in the analysis of electroencephalography (EEG) data.

2.2 Working Memory and Code Review

For code review, there is evidence of the influence of expertise on effectiveness (*e.g.*, McIntosh et al (2015)), but no studies on the influence of working memory capacity on code review performance. Hungerford et al (2004) studied cognitive processes in reviewing design diagrams and observed different strategies with varying performance. In a think-aloud protocol analysis study, Robbins and Carver (2009) 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 (2009) found that more structured techniques lead to higher cognition levels. Recent work by Ebert et al (2017) tries to measure signs of confusion from remarks written by the reviewers. The model of human and computer as a joint cognitive system (Dowell and Long, 1998), also called distributed cognition, has been proposed by Walenstein to study cognitive load in software development tools (Walenstein, 2003, 2002).

Given the importance of the reviewer's abilities in code reviews, the lack of research on the influence of working memory on reviews, and the existence of promising results in the area of natural language understanding, associating working memory capacity and review performance seems like a promising field to study.

2.3 Code Change Size/Untangling in Code Review

One likely determinant of a change-based review's mental load is the code change's size and complexity. The hypothesis that large, complex changes are detrimental to code review performance is reflected in many publications, for example MacLeod et al.'s guideline to "aim for small, incremental changes" (MacLeod et al, 2017), which can be found similarly also in the earlier works by Rigby et al. (Rigby et al, 2012) and Rigby and Bird (Rigby and Bird, 2013). These guidelines are generally based on interviews with developers and observations of real-world practices. Similar guidelines also exist for

code inspection (*e.g.*, by Gilb and Graham (1993)). In the context of design inspection in industry, Raz and Yaung (1997) found support for higher detection effectiveness for smaller review workloads. Rigby et al (2014) build regression models for review outcomes based on data from open-source projects. They predict review interval (*i.e.*, time from publication of review request to end of review) and the number of found defects. The nature of their data prevents them from building models for review efficiency and effectiveness. They observe effects for 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 the light of our hypotheses, the latter observation should be the result of two confounded and opposing effects: A higher total number of defects and a smaller review effectiveness in larger changes.

Change untangling refers to splitting a large change that consists of several unrelated smaller changes into these smaller changes, in line with reducing the mental load to review each change. It has first been studied by Herzig and Zeller (2013) and alternative approaches have been proposed by Dias et al (2015), Platz et al (2016), and Matsuda et al (2015). Barnett et al (2015) investigated change untangling in the context of code review and obtained positive results in a user study. Tao and Kim (2015) also proposed to use change untangling for review and showed in a user study that untangling code changes can improve code understanding.

Summing up, many studies and guidelines for practitioners build on the hypothesis that reducing change size and complexity is worthwhile. As we are not aware of detailed data on the influence of change complexity on change-based code review performance from controlled settings, we investigate it in our study.

2.4 Ordering of Code Change Parts for Review, Reading Techniques, and Delocalized Defects

When a reviewer gets in touch with a code change, he or she has to read it in a certain order. Intuitively, an order is good when it helps the reviewer to build up an understanding of the code change in a structured and incremental fashion. On the other hand, an order that seems random and in which the flow between related change parts is disturbed by other change parts might hinder understanding. In this regard, Baum et al (2017b, pg. 8) argue: "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. ... A good order [of the way in which the change parts are presented] helps [code change] understanding by reducing the reviewer's cognitive load and by an improved alignment with human cognitive processes ... . It helps checking for defects by avoiding speculative assumptions and by easing the spotting of inconsistencies."

They present six principles for a good order (Baum et al, 2017b, pg. 5f):

1. Group related change parts as closely as possible.

2. Provide information before it is needed.
3. In case of conflicts between Principles 1 and 2 prefer Principle 1 (grouping).
4. Closely related change parts form chunks treated as elementary for further grouping and ordering.
5. The closest distance between two change parts is "visible on the screen at the same time."
6. To satisfy the other principles, use rules that the reviewer can understand. Support this by making the grouping explicit to the reviewer.

They proceed by formalizing these principles and define a relation $\geq_T$ ('is better than') among orders of the change parts (called 'tours') and claim that when one order is better than another, it will lead to higher (or sometimes equal) review efficiency (Baum et al, 2017b, pg. 7f). As an example, assume there are three change parts a, b, and c, of which a and b are related by similarity. Then it holds that $(a, b, c) \geq_T (a, c, b)$ and also that $(c, a, b) \geq_T (a, c, b)$. The tours (a, b, c) and (c, a, b) are incomparable with regard to $\geq_T$, *i.e.*, the theory does not specify which of them is better. This ordering theory was derived based on data from observations and surveys, but has not been tested directly so far.

Geffen and Maoz tested the influence of method ordering inside a class on time for and correctness of understanding. They observe among other things a tendency that putting methods related by call-flow together increases efficiency of understanding, especially for less experienced participants, but their results did not reach statistical significance (Geffen and Maoz, 2016). Another study of code ordering, but without the inclusion of a controlled experiment, has been performed by Biegel et al (2012). Both of these studies are rather exploratory and lack our theoretical underpinning regarding optimal code ordering. All in all, there is currently a lack of firm empirical support that the above-mentioned ordering principles will improve review performance. Implementing the principles in software could be an easy way to improve the performance of tool-assisted change-based code reviews, which poses the need for this empirical support.

Other aspects of code review have been subject to more intense experimentation, often in the context of classic Inspection. The use of reading techniques to improve code review shares some similarities with change part ordering, as a reading technique often prescribes a certain order of reading the source code. Basili et al. summarize many early results on reading techniques (Basili et al, 1996). The theoretical background of many reading techniques differs from ours in putting emphasis on forcing the reviewer into an active role and not on reducing its mental load.

Dunsmore et al (2000, 2001, 2003) put forward 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. They developed a reading technique to cope with these delocalized defects and tested it in a series of experiments.

Table 1 Summary of Important Terminology/Constructs.

Construct	Description
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'. (Wilhelm et al, 2013)
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. (Paas and Van Merriënboer, 1994)
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 that depends only on the task or environment, *i.e.*, which is independent of subject characteristics. (Paas and Van Merriënboer, 1994)
Code Change	The 'code change' consists of all changes to source files performed in the unit of work (Baum et al, 2016a) under review. The code change defines the scope of the review, *i.e.*, the parts of the code base that shall be reviewed. (Baum et al, 2017b)
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. (Baum et al, 2017b) In the context of our study, we combined hunks from the same method into one change part.
Tour	A 'tour' is a sequence (permutation) of all change parts of a code change. (Baum et al, 2017b) We also use 'code change part order' as a synonym.
Delocalized Defect	A defect that can only be found or found much more easily by combining knowledge about several parts of the source code. (Dunsmore et al, 2001)
Review Efficiency	'Review efficiency' is the number of defects found per review hour invested. (Biffl, 2000)
Review Effectiveness	'Review effectiveness' is the ratio of defects found to all defects in the code change. (Biffl, 2000)
Review Performance	In this article, we consider 'review performance' to consist of review efficiency and review effectiveness.

The order of reading that follows from their abstraction-based reading technique shares some similarities to that of Baum et al (2017b). They did not find a significant influence of their technique on review effectiveness and did not analyze review efficiency. When it was later analyzed in an experiment by Abdelnabi et al (2004), a positive effect on efficiency could be found, and the results of an experiment by Skoglund and Kjellgren (2004) are also largely compatible with these findings.

As announced, Table 1 summarizes important terminology introduced in this section.

3 Experimental Design

In the following, we detail the design of our experiment.

3.1 Research Questions and Hypotheses

We structure our work along three research questions, which we introduce in the following. After each research question, we formalize it as null and alternative hypotheses.

Based on the importance of human factors for improving code review performance (Sauer et al, 2000), prior research in text and hypertext comprehension that shows an influence of working memory capacity on comprehension (DeStefano and LeFevre, 2007), and the aforementioned hypotheses by Baum et al (2017b), we ask:

> **RQ$_1$.** 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 (Hungerford et al, 2004; Robbins and Carver, 2009) and we hypothesize that higher working memory capacity is beneficial in doing so. Therefore, we look for differences in the number of defects found. We also analyze the subset of delocalized defects. We have the following null and alternative hypotheses:

- $H_{1.1.0}$ There is no positive 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 positive 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.

Earlier work brought evidence that large, complex[3] code changes are detrimental to review performance, albeit this evidence is mostly based on qualitative or observational data (MacLeod et al, 2017; Raz and Yaung, 1997; Rigby and Storey, 2011). Because much work builds upon this hypothesis, we report on more reliable support and quantitative data from controlled settings for it:

> **RQ$_2$.** Does higher mental load in the form of larger, more complex code changes lead to lower code review effectiveness?

Specifically, we hypothesize 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 lead to demotivation and faster review (skimming), but it may also lead to longer

[3] Size and complexity are often highly correlated (Hassan, 2009), therefore, we do not treat them separately in the current article.

review times; consequently, we only test for the effect of code change size on effectiveness formally.

The probability of detection can vary greatly between different defect types. Therefore, we select one particular defect type for this RQ: the swapping of parameters in a method call, a special kind of delocalized defect. Among the defects seeded into the code changes, one defect in the small code change (Swap_{WU}) and two defects in one of the large code changes (Swap_{S1} and Swap_{S2}) are such swapping defects. The corresponding null and alternative 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 probabilities for Swap_{WU} and Swap_d differ 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', we have four null and four alternative hypotheses.

There are numerous possibilities to reduce the cognitive load during code review, *e.g.*, change untangling or detection of systematic changes (Zhang et al, 2015). In our recent theory, we argue that optimizing the order of presenting the code is another such possibility (Baum et al, 2017b). We test this claim:

RQ_3. 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 our theory predicts that review efficiency and effectiveness for a is not inferior to that of b, and it also predicts that there are cases where efficiency is greater. Focusing on the latter, this leads to the following null and alternative hypotheses:[4]

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

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

For a given code change of non-trivial size, there is a vast number of possible permutations and consequently many possible comparisons to perform; we have to select a subset for our study. As this study is the first to measure the effect of change part ordering on code review, our choice is exploratory: We compare 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. To also include a more realistic comparison, we select one of the best and one of the worst orders that keep change parts from the same file together. In the following, these order types will be called as described by column *ID* in Table 2.

By construction, it holds that $OF \geq_T WF$, $ONF \geq_T WNF$, $WF \geq_T WNF$ and, by transitivity, $OF \geq_T WNF$. We consider the first 3 pairs.

[4] we use the two-sided formulation for reasons of conservatism, even though the theory's prediction is one-sided

Table 2 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

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: $H_{3.OF.0}$, $H_{3.OF.A}$, $H_{3.ONF.0}$, ...

3.2 Design

Fig. 1 shows an overview of the phases, participation, and overall flow of the experiment that we designed to answer our research questions and test the aforementioned hypotheses. The general structure is that of a partially counter-balanced repeated measures design (Field and Hole, 2002), augmented with some additional phases. Here we briefly describe each phase, which we are going to detail in the rest of this section.

(1) The experiment is entirely done through an instrumented browser-based tool that allows 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" of participants in software engineering experiments (Falessi et al, 2017).
After filling this information, the participant receives more details on the tasks and the expectations regarding their reviews. Moreover, the participant is shown a brief overview of the relevant parts of the open-source software (OSS) project from which we sampled the code changes to review.[5]

[5] All these descriptions could be accessed again on demand by participants during the review.

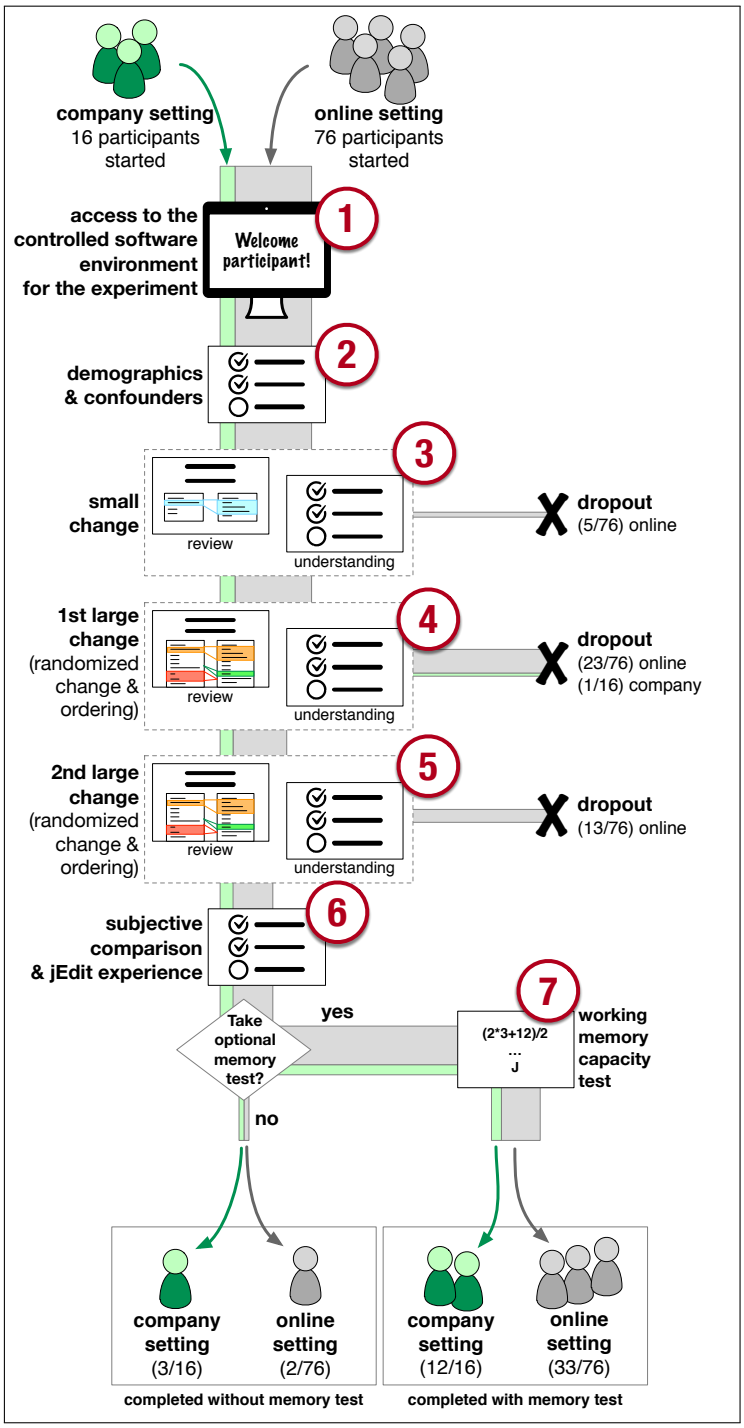

Fig. 1 Experiment steps, flow, and participation

(3) Each participant is then asked to perform a review on a small change; afterward the participant has to answer a few understanding questions[6] on the code change just reviewed. This phase is needed both for answering RQ_2 and as a *warm-up review*. As a warm-up, this review is proposed to mitigate the effects of the novelty of the code base and the review UI in the next two large reviews.
(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 the randomly selected ordering type (see Section 3.4) for answering RQ_3. Similarly to 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, which was ordered according to the second sampled ordering type.
(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, we ask for the participant's experience with the OSS system from which we sampled 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 (Unsworth et al, 2005), based on the shortened operation span test by Oswald et al (2015), which we re-implemented for our browser-based setting. We use this task to measure the working memory capacity for answering RQ_1. This task is optional because (1) working memory capacity as a component of general intelligence is more sensitive data than most other data we collect, thus participants should be able to partially opt-out, and (2) the test can be quite tiring, especially after having completed a series of non-trivial code reviews.

3.3 Browser-Based Experiment Platform

We devised 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 allowed us to ease access to professional developers and to control the ensuing threats to validity. In the next paragraph, we detail the part we 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

[6] The full text of these questions is contained in the replication package (Baum et al, 2018).

Fig. 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).

was written to file as log records, which were then downloaded and analyzed offline.

In current industrial practice, browser-based code review tools that present a code change as a series of two-pane diffs are widely used.[7] Therefore, we implemented a similar UI for our study. Although this setup allows for free scrolling and searching over the whole code change and therefore introduces some hardly controllable factors into our design, we chose it in favor of more restrictive setups because of its higher ecological validity. During code review,

[7] *e.g.*, GitHub pull requests, Gerrit and Atlassian Stash/Bitbucket

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 Fig. 2. The HTML page for a review starts with a description of the commit. These descriptions were taken from the original commit messages, slightly adapted for clarity. 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 a maximum of 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.

3.4 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, we decided to use one that is probably little known to all participants. This increases the difficulty of performing the reviews. To keep the task manageable, we ensure that at least the functional domain and requirements are well known to the participants and that there is little reliance on special technologies or libraries. To satisfy these goals, we selected the jEdit programmer's text editor[8] as the basis from which to select code changes to review, as others have done before us (Rothlisberger et al, 2012). To select suitable code changes, we screened the commits to jEdit from the years 2010 to 2017. We programmatically selected changes with a file count similar to the median size of commercial code reviews identified in previous work (Barnett et al, 2015; Baum et al, 2017b). 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". The selected commits are those of revision 19705 ('code change A' in the following) and of revision 19386 ('code change B'). In addition, we selected a smaller commit (revision 23970) for the warm-up task. Code change A is a refactoring, changing the implementation of various file 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 3. 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.

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

Table 3 Code Change Sizes 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]	Total Defects	Delocalized Defects
Small / Warm-up	2	3	31	3	1
Large code change A	7	15	490	9	2
Large code change B	7	21	233	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

To reduce the risk of some participants looking for further information about jEdit (*e.g.*, bug reports) on the internet while performing the experiment, we removed all mentions of jEdit and its contributors from the code changes presented to the participants and only gave credit to jEdit after the end of all reviews. We 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 (Bacchelli and Bird, 2013; Baum et al, 2016b), with detection of faults (correctness defects; (IEEE 24765, 2010)), improvement of code quality (finding of maintainability issues), and spreading of knowledge often among the most important ones. As the definition of maintainability is fuzzy and less consensual than that of a correctness defect, we restrict most parts of our analysis 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, we seeded several further defects, 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, from rather simple ones (*e.g.*, misspelled messages and forgetting a boundary check) to hard ones (*e.g.*, a potential stall of multithreaded code and a forgotten adjustment of a variable in a complex algorithm). We regard 6 of these defects as delocalized (Dunsmore et al, 2000), *i.e.*, their detection is likely based on combining knowledge of different parts of the code. The first and last authors independently classified defects as (de)localized and contrasted results afterward. Both types of defects can be seen in Fig. 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, whereas 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 (Baum et al, 2018).

Measurement of Defects. We explicitly asked the participants to review only for correctness defects. All review remarks were manually coded by one

of the authors, 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, we decided to count a remark if it was in the right position and could make a hypothetical author aware of his defect, whereas 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). The detailed coding of all review remarks can be seen in the study's online material (Baum et al, 2018). 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 coded a subset of the remarks and discrepancies were discussed.

Code order. 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 (2000)). To gain maximum information from our 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 decided to restrict ourselves to pairs of change part order types, specifically, those pairs needed for checking the predictions for RQ_3: 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, we use 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, we first split the code changes into change parts. We 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, we determined the relations between the change parts. We checked for the following subset of the relation types given by Baum et al (2017c) (manually, except for Jaccard similarity): (1) Similarity (moved code or Jaccard similarity (Jaccard, 1912) of used tokens > 0.7), (2) declare & use, (3) class hierarchy, (4) call flow, and (5) file order. Our previous work did not specify which relation types should be regarded as more important than others. To determine a concrete order, we 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, we implemented the $\geq_T$ relation in software[9] and semi-automatically determined minimal and maximal elements of this partial order relation.

Time. One of our 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 (Gilb and Graham,

[9] available as part of CoRT: https://github.com/tobiasbaum/reviewtool

1993), which lets many researchers control for time by fixing it to a certain amount. This would run contrary to one of our main research goals, *i.e.*, finding differences in efficiency. Therefore, we chose to let participants review as long as they deem it necessary and measure the total time. A participant who needed to interrupt the review could press a 'pause' button; 14 participants did so at least once. We measure gross time (including pauses) and net time (without pauses).

Working Memory. As described above, working memory capacity can be measured with complex span tests (Daneman and Carpenter, 1980) that consist of interleaved time-limited recall and cognitive processing tasks. Our implementation of the shortened automated operation span test (Unsworth et al, 2005; Oswald et al, 2015) consists of two tasks each with 3 to 7 letters that are shown for a brief amount of time and have to be remembered while solving simple arithmetic tasks after each letter (see the online material for more details (Baum et al, 2018)). Each letter remembered correctly gives one point, so the maximum score is 50. The theoretical minimum is zero, but this is quite unlikely for our population. Rescaling the results from Oswald et al (2015) 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.

3.5 Data Analysis

We employ various statistical procedures and tests to answer our research questions.

For RQ_1, we look for correlation using Kendall's τ_B correlation coefficient (Agresti, 2010). We chose τ_B because it does not require normality and can cope with ties, which are likely, since we deal with defect counts.

For RQ_2, we have 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), we use McNemar's exact test (Agresti, 2007).

We performed power analysis for RQ_3, because we expected a smaller effect for it and we had more groups compared to the other RQs. We estimated effect sizes and some parameters and performed randomized simulation runs to test several possible methods in this way. To deal with our incomplete repeated measures design with potentially imbalanced groups and several confounding factors, we planned to use linear mixed effect (lme) regression models (Bates et al, 2014) and determine confidence intervals for the coefficients using parameterized bootstrap. It turned out that some of our assumptions in the simulation were wrong, and the empirical data does not satisfy all assumptions needed for lme models. An alternative is the Wilcoxon signed-rank test, but it is also problematic due to imbalanced groups. Therefore, we will present both results to the reader. The R code of all analyses is available (Baum et al, 2018).

Table 4 The Variables Collected and Investigated for this Study.

Independent variables (design):	
Working memory span score (measured)	ordinal/interval
Used change part order type (controlled)	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 tiredness 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

Whether and how to correct for alpha errors when testing several hypotheses is disputed in some research communities (*e.g.*, Perneger (1998)). We decided to perform Bonferroni-Holm correction within research questions and no correction spanning several research questions. Therefore, the probability of a type I error should be at the nominal 5% for each research question; the error probability for the paper as a whole is larger. When we give confidence intervals, those are 95% intervals.

3.6 Participants

The power analysis for RQ_3 indicated a need for at least 60 participants. Furthermore, to see an effect of change part order, we needed code changes larger than a minimum size. Combined, this meant that it is infeasible to use a code base that is well known to all participants and that inexperienced participants (*e.g.*, students) would be overwhelmed by the task. Our 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
– how often they program.

The experiment UI was made available online in 2017 for six weeks. Similar to *canary releasing* (**?**), we 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 used the authors' personal and professional networks and spread the invitation widely over mailing lists, Twitter, Xing, Meetup and advertised on Bing. We convinced the complete development team of a collaborating software development company to participate 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 our sampling method is that we could not assign participants to groups in bulk before the experiment, but had to assign them with an online algorithm. To perform the assignment, we used a combination of balancing over treatment groups, minimization based on the number of defects found in the small change review (Point 3 in Fig. 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 decided to offer financial compensation. We did this in form of a lottery (Singer and Ye, 2013; Laguilles et al, 2011), 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, participants who did not complete all reviews will not be taken into account in further analyses. We regard participants who spent less than 5 minutes on a review and entered no review remark as 'did not finish'. We also excluded one participant who 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. 4 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 1 has none. Fig. 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 our sample is about 3.7 points higher than estimated based on the sample of Oswald et al

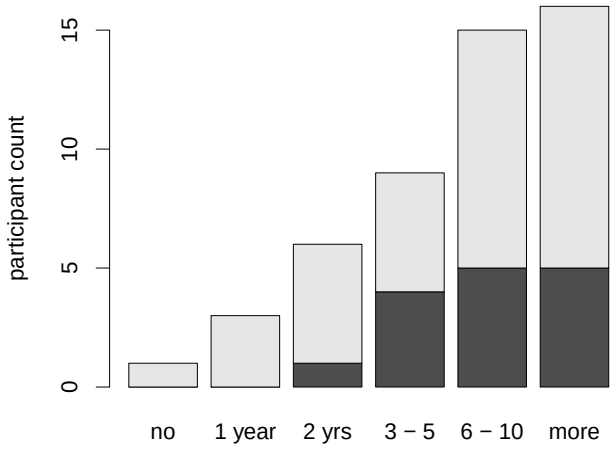

Fig. 3 Professional development experience of the 50 participants that finished all reviews. Darker shade indicates company setting, lighter shade is pure online setting.

(2015) (from a different population), and there seems to be a slight ceiling effect that we will discuss in the threats to validity.

4 Results

In this section, we present our empirical results and the statistical tests performed. Before that, we briefly describe general results on the participant's performance and a brief analysis of their qualitative remarks. The complete dataset is available (Baum et al, 2018).

Table 5 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 quite large ordering/fatigue effect, particularly striking for the drop in review time between the first and second large review. We discuss this ordering effect in Section 5.1.

We analyzed the participant's full-text answers for potential problems with the experiment. Most comments in this regard revolved around some of the

Table 5 Mean and Standard Deviation (sd) for the Number of Defects Found and Review Time Depending on Review Number and Code Change

	Defects	Time
Small / Warm-up	1.76 (of 3), sd=1.13	8.49 min., sd=5.48
First large review	3.68, sd=2.12	30.03 min., sd=16.41
Second large review	2.98, sd=2.16	20.69 min., sd=17.32
Large code change A	3.16 (of 9), sd=2.09	23.85 min., sd=16.96
Large code change B	3.5 (of 10), sd=2.23	26.88 min., sd=17.93

compromises we settled for 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"$_{PC11}$[10]), 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!"$_{PO37}$).

We also scanned the participants' scrolling and UI interaction patterns. These patterns indicate that some participants made quite 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 (Baum et al, 2018).

4.1 RQ$_1$: Working Memory and Defects Found

For RQ$_1$, our goal is to analyze the relationship between working memory span score and number of defects found. As motivated in Section 3.1, we analyze all defects as well as delocalized defects. We use Kendall's τ_B rank correlation coefficient (one-sided) to check whether a positive correlation exists. For all defects, τ_B is 0.05 (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, we can reject the null hypothesis $H_{1.2.0}$ for delocalized defects, yet with a rather small Kendall correlation, and cannot reject the null hypothesis $H_{1.1.0}$ for all defects.

Looking at the scatter plot in Fig. 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, or put differently, 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 one. We scrutinized the three most influential participants both for and against the hypothesis, checking the time taken, found defects, answers to understanding questions and other measures. Based on these in-depth checks, we decided to stick to the formal exclusion criteria described in Section 3.6 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.

[10] The subscripts behind the citations are participant IDs, with POn from the online setting and PCn from the more controlled setting.

Fig. 4 Scatter plot of Working Memory Span and Number of Delocalized Defects Detected; Slight jitter added

Table 6 Working Memory Capacity and Confounding Factors with the Most Significant Correlations, Information Whether the Factor is Contained in the Optimal Linear Regression Model Found by Stepwise BIC for Delocalized (deloc.) or All Defects, and Correlation Between Factor and Working Memory Span (w.mem.).

Factor	τ_B all defects	τ_B deloc. defects	stepw. BIC	τ_B w.mem.
Working memory span	0.05	0.24	deloc. only	–
Review time	0.35	0.44	both	0.07
Review practice	0.2	0.31	deloc. only	-0.01
Controlled setting	0.28	0.3	deloc. only	-0.18
Prof. dev. experience	0.24	0.24	deloc. only	0.04

To investigate the other influencing factors, we used stepwise BIC (Bayesian Information Criterion) (Venables and Ripley, 2002) to fit a regression model. Additionally, we determined the correlation between all factors and the defect counts. The only further effect strong enough for inclusion in the regression model for all defects was *review time*, with longer reviewing time leading to more defects found. Of the remaining factors, professional development experience and review practice also seem to be positively correlated with the total number of defects found, as is the experimental setting. The other factors, *e.g.*, subjective mental freshness, have a lesser influence and did not make it into the regression model. Table 6 gives details on the confounders.

RQ$_1$: *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.*

Table 7 Count of reviews in which the respective defects were detected and p-value from McNemar's test for RQ_2

	Defect found in review				
Hypothesis	small only	large only	both	none	p
$H_{2.S1.first\ large\ review.0}$	10	0	9	6	0.002
$H_{2.S2.first\ large\ review.0}$	11	0	8	6	0.001
$H_{2.S1.second\ large\ review.0}$	10	1	9	5	0.0117
$H_{2.S2.second\ large\ review.0}$	12	1	7	5	0.0034

4.2 RQ_2: Code Change Size and Defects Found

In RQ_2, we want to test whether there is a difference in review performance between small vs. larger, more complex code changes. It could already be seen in Table 5 that the performance of the small review was higher than that of the larger reviews. In numbers, 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 reviews and 9.47 defects/hour for the large reviews. These numbers depend to a large degree on the seeded defects. For a fair comparison, we picked (as described in Section 3.1) a specific defect type to compare in detail: The swapping of parameters 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 this and the other situations can be seen in Table 7. The corresponding tests for association are all highly statistically significant (the smallest threshold, after applying Bonferroni-Holm correction, is 0.05/4=0.0125). Therefore, we can reject all four null hypotheses $H_{2.<d>.<n>.0}$ and conclude that there is a statistically (and practically) significant difference in the number of 'swap type' defects found between small and large code changes. Contrary to the large effect when comparing the small and large code change, the difference between the first and second large review, *i.e.*, a fatigue or other ordering effect, is nearly non-existent for this defect type.

Part of the hypothesis underlying RQ_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 we would expect a higher drop in detection effectiveness for them. We checked for such an effect, but it is far from statistically significant. All in all, we cannot 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$_2$: *Larger, more complex code changes lead to lower code review effectiveness. This may be caused by higher mental load or by other reasons, such as faster review rates or lower motivation.*

4.3 RQ$_3$: Code Order and Review Efficiency

In RQ$_3$, we analyze whether there is a difference in review efficiency depending on the order of the code.

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 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).

Fig. 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 9 shows the exact numbers). As mentioned in Section 3.5, we originally planned to analyze the data with a linear mixed effect model (Bates et al, 2014), because it is better able to take the slightly unbalanced and incomplete nature of our data into account, but its preconditions (homoscedasticity, normality of residuals) were not fully satisfied. Therefore, we also analyzed the data 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, whereas for ONF vs WNF there is a conflict between mean and median and the effect is less clear. 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

Table 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 is first, OF+Change B is 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.

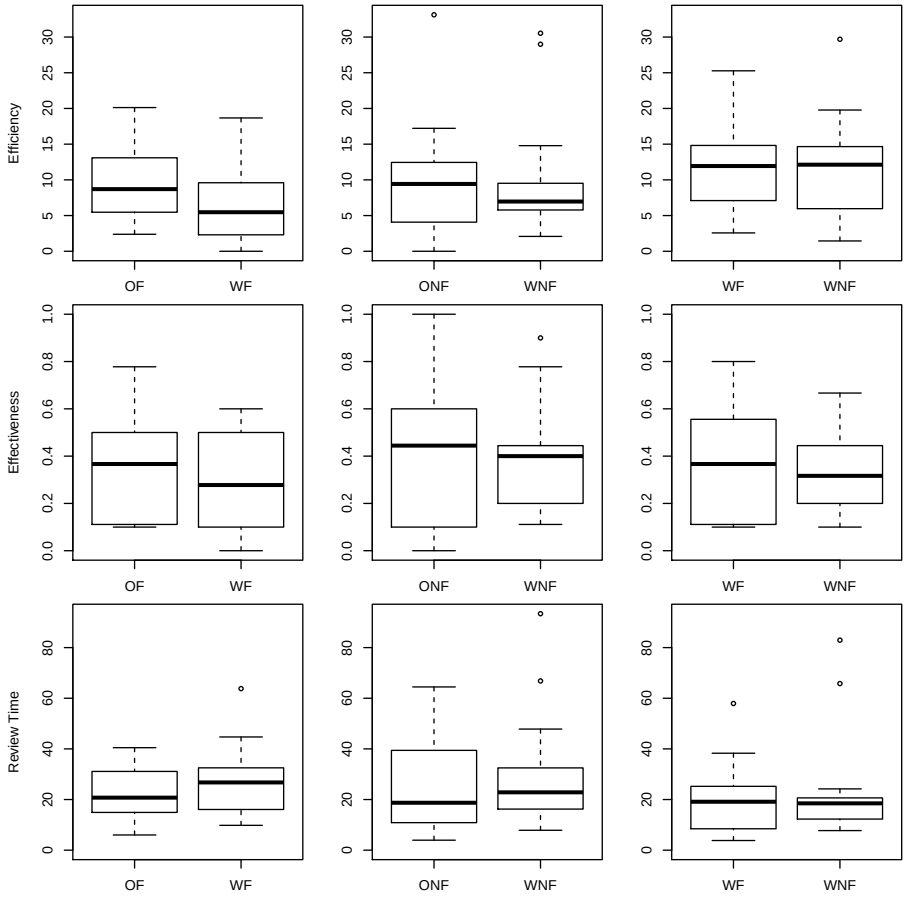

Fig. 5 Box plots for review efficiency (in defects/hour), effectiveness (found defects/total defects), and review time (in minutes) for the three treatment groups. In each plot, the left treatment is the theoretically better one.

lesser working memory capacity. Except for the ONF-WNF sub-sample, the tendencies for the respective sub-samples in Table 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 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_3 is that a better order means less mental load. To roughly assess mental load, we asked participants to rate which of the large changes was subjectively more complicated. In line with our hypothesis, a majority of the participants rated the worse order as more complicated (answer as expected: 26 participants, no difference: 11, inverse:

Table 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, we left out lmer models for low wm span and their intervals are inaccurate. Every row from the upper part is continued in the lower part. "conf.int." = "confidence interval", "sd" = "standard deviation"

		theoretically better treatment		theoretically worse treatment	
group	n	median (conf.int.)	mean (sd)	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)

	comparison			
group	relative difference of medians	p (Wilcoxon)	Cohen's d	lmer coeff. (conf. int.)
all	44%	0.2587	0.13 (negl.)	
low wm span	67%	0.2479	0.18 (negl.)	
OF-WF	59%	0.1084	0.43 (small)	2.66 (-0.67 .. 5.7)
OF-WF, low	91%	0.0537	0.62 (medium)	–
ONF-WNF	35%	1.0000	-0.07 (negl.)	-0.36 (-2.57 .. 1.94)
ONF-WNF, low	21%	0.5625	-0.38 (small)	–
WF-WNF	-2%	0.6698	-0.06 (negl.)	-0.38 (-3.76 .. 3.17)
WF-WNF, low	1%	0.6250	-0.07 (negl.)	–

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 by Baum et al. (Baum et al, 2017b) 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 Fig. 5

and the confidence intervals in Table 9 gives little reason to expect statistically significant support or rejection, we do not perform formal non-inferiority tests.

> **RQ$_3$**: *Strong statistically significant conclusions cannot be drawn. An effect of better change part ordering on review efficiency may exist, but its strength is highly dependent on the respective orders. The tendencies are generally compatible with the predictions by Baum et al (2017b), but there is no statistical confirmation.*

5 Discussion

We now review the validity of our design and the limitations that emerged. We also discuss our main findings and their implications for further studies and tools, and the lessons that we have learned while conducting this study.

5.1 Validity and Limitations

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 our study. There is a risk of the participants not being as motivated as they are in real code reviews, which we tried to counter by making code review efficiency part of the precondition to winning the cash prize. External validity is mainly hampered by three compromises: Usually, code reviews are performed for a known code-base, the defect density in the industry is usually lower than in our code changes, and we asked participants to mainly review for correctness defects, although identification of maintainability defects is normally an important aspect of code review (Mantyla and Lassenius, 2009; Thongtanunam et al, 2015a). This could pose a threat to external validity if the mechanisms for finding other types of defects are notably different. We believe this is not the case, 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 worthwile to investigate 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, we employed a multi-stage process: After tests among the authors, we performed three pre-tests with one external participant each, afterward we started the canary release phase.

Many of the constructs we use are defined in previous publications and we reuse 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. We did not formally check whether our 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 (Wilhelm et al, 2013). Compared to working memory, the mental load construct is less well defined and usually assessed using subjective rankings. For RQ_3, we use a simple subjective ranking; for RQ_2 we assert that code changes which are an order of magnitude larger will correspond to higher mental load, without measuring it explicitly. Therefore, we can assess mental load only qualitatively and not quantitatively in the current article.

One of the central measures in our study is the number of defects found, which we restricted to correctness defects to avoid problems with fuzzy construct definitions. To reduce the risk of overlooking defects that a participant has spotted, we asked participants to favor mentioning an issue when they are not fully sure if it really is a defect. We regard this advice as compatible with good practices in industrial review settings (Gilb and Graham, 1993). The defects were seeded by the first author, based on his 11-years experience in professional software development and checked for realism by another. Nevertheless, we cannot rule out implicit bias in seeding the defects as well as in selecting the code changes.

We advertised our experiment worldwide. Nonetheless, we made our experimental materials available in English only. We deem this acceptable as English reading skills are common among software developers and large parts of the experiment consisted of reading source code and not text. We accepted review remarks given in other languages.

Another of the central measures is the time taken for a review. By having a 'pause' button and measuring time with and without pauses, we allowed participants to measure time more accurately, but we cannot assure 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, we did not tell participants 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. We asked participants to not talk to others about details of the experiment but cannot tell whether everyone 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 sample of 50 professional software developers is quite large in comparison to many experiments in software engineering (Sjøberg et al, 2005). For other sources of variation, we had to limit ourselves to considerably smaller samples, leading to a risk of mono-operation bias and limited generalizability: We only have three different code changes, only four different change part orders for RQ_3 and analyzed only one defect type in detail for RQ_2.

A common threat in software engineering studies is hypothesis guessing by the participants. In our study, we stated only abstractly that we are interested in improving the efficiency of code review and did not mention ordering of

code at all. Furthermore, we put more risky parts with regard to hypothesis guessing, such as the working memory span test, at the end.

Internal Validity. A likely consequence of the difficult task is the ordering effect we observed, *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_3 we tried to counter the ensuing risk, but due to drop-outs and the failed assumptions of the lme model we did not fully succeed. We observed a quite high drop-out rate, likely again a consequence of the difficult task and the online setting. A slightly larger share of drop-outs, 23 of 37, happened when either a WF or WNF order was shown. On average, the drop-outs had lower review practice and performed less well in the short, warm-up review (differential attrition), which could partly explain the differences between groups described in Section 4.3. We had to analyze each treatment group separately to counter that risk.

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, we included questions to characterize our sample (*e.g.*, experience, role, screen size). To identify and exclude duplicate participation, we logged hashes of participant's local and remote IP addresses and set cookies in the browser. By having a subset of the participants perform the experiment in a controlled setting, we controlled this threat further by comparing the two sub-samples. Again, the differences in overall efficiency and effectiveness are not significant, but 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. We asked the participants to review in full-screen mode and did not mention jEdit, but we cannot fully rule out that participants in the online setting searched for parts of the code on the internet. If somebody did, this would increase the noise in our data. Because we had to use an online algorithm to assign participants to treatment groups, we could not create groups as balanced as in a setting with a set of participants known in advance.

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. We do not believe this poses a major risk to the validity of our main findings; on the contrary, we would expect stronger effects with a more representative sample. The working memory span scores we observed were higher than those observed in other studies (Unsworth et al, 2005; Oswald et al, 2015). The resulting slight ceiling effect might have reduced statistical power in the analyses including working memory span scores. We attribute this effect to the selection bias[11] and possibly also to a general difference in working memory span scores between software developers and the general population, but it could also be a sign of a flaw in our implementation of the working memory span test. A downside

[11] Indicated by the negative correlation between company/online setting and working memory (Table 6)

of having the working memory span test at the end is that we 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, we would like to show a causal relation between working memory capacity and review effectiveness for RQ_1. This demands controlled changes to working memory capacity (Pearl, 2001), which is ethically infeasible. Therefore, we check for potential confounders (see Table 4) but cannot reliably rule out unobserved confounders and report only on associations.

We randomized the order of the two large reviews, but the small review was always first. This is owed to its dual role as a warm-up review for RQ_3 and as part of RQ_2. We chose this compromise because there already exists evidence for RQ_2. By randomizing the order of the large reviews and having a warm-up review, we avoid threats due to ordering, maturation and similar effects for RQ_3.

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 our power analysis for RQ_3, 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 we did not reach statistical significance for RQ_3. Further threats to conclusion validity for RQ_3 are the only partially satisfied assumptions for the mixed effect regression models and the slightly unbalanced assignment to treatment groups. All in all, we are not able to draw reliable conclusions for RQ_3.

5.2 Implications and Future Work

The results of the experiment are generally compatible with earlier results and hypotheses: (1) There is a possibly mediated influence of working memory capacity on certain aspects of software development performance (in our case: finding of delocalized defects); (2) code change size is indeed a major factor influencing code review performance; (3) on the predictions of our proposed ordering theory, we found largely compatible tendencies, but no statistically significant results. A possible contradiction to the theory is the comparison of means for the condition that does not respect file boundaries (ONF-WNF). Future studies can be designed to investigate further whether this is a statistical artifact or does indeed point to a problem in our theory, in addition to replicating to gain higher power and more reliable results.

Working memory in review studies and tools. Given the evidence that working memory capacity (1) does influence review performance for delocalized defects and (2) can be measured computerized in around 10 minutes per participant, it is reasonable to recommend researchers to measure it as a

potential confounding factor. This finding also has practical implications for reviewer recommendation (Thongtanunam et al, 2015b): 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 (Baum and Schneider, 2016).

Natural-born reviewer? Not exclusively. Working memory capacity influences only the detection of certain defect types, moreover review performance is influenced by other factors like the time taken and (possibly) experience and practice. This suggests that—to a large degree—*one is not born as a good reviewer, but one learns to act as one.* This finding is a compelling argument to conduct research on how to help developers become good reviewers faster. Furthermore, there is value in helping reviewers with lower working memory span to overcome their limitations. Better change part ordering is just one potential avenue; another example is to provide summarizations of the change (Pollock et al, 2009; Baum and Schneider, 2016) to help the reviewer overcome working memory limits by chunking. 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 are there at all? Studies can be designed and carried out to investigate better review support for defect types that are currently not found easily. The higher difficulty of finding delocalized defects also underlines the validity of common software architecture guidelines like encapsulation and cohesive modules (Parnas, 1972), and might also hint to the benefits of good software design for reviews.

Change features and review performance. The confirmation that review performance declines for larger, more complex code changes strengthens the case for the research on change untangling and identification of systematic changes. We did not explore the underlying mechanisms in much detail, therefore open questions remain: Is the change in review performance more an effect of complexity, as suggested by the hypothesis on mental load, or an effect of size? What happens when smaller code changes can only be reached by an "unnatural" division of the work? The mean efficiency and effectiveness for code change A and code change B are astonishingly similar given their differing sizes and characteristics. Were we just lucky when trying to select code changes of similar complexity, or is there some kind of saturation effect? Reducing emphasis on external validity in a follow-up study would allow the construction of artificial code changes to analyze these questions systematically. Another notable property of our data is related to the observed ordering effect: Review times were much shorter for the second large review and mean review effectiveness is smaller, as expected. But review efficiency actually increased. Studies could investigate the counter-intuitive inference that it could be more efficient to review more superficially, perhaps with Rasmussen's

model of rule-based versus knowledge-based cognitive processing as a theoretical background (Rasmussen, 1983).

Code change ordering. Albeit we did not reach statistical significance for the effect of change part ordering, we regard the results as interesting enough to support further effort put into code review tooling that sorts change parts into an improved order. Once such tooling is available, the risk and effort of using it in a team are very small, compared to classic reading techniques that have to be taught to every reviewer. To provide more convincing support for the underlying hypothesis that the effect is caused by reduced cognitive load, cognitive load could be measured explicitly (Chen et al, 2016; Fritz et al, 2014). Further insights could also be gained by more directly studying the processes in the brain (Siegmund et al, 2017). An alternative approach to find good orders for reviewing code could be to use empirical data from successful reviews to learn models of navigation behavior.

Cognitive load reduction as a paradigm in review research. Is the model of mental and cognitive load adequate to explain performance in code reviews? Our results generally support the model, but it could be too simple and abstract to be useful to explain many of the above-listed effects and to guide research. One example we would like to see improved are the currently quite fuzzy definitions of the central constructs of mental and cognitive load. Research can be carried out to investigate an extended version of the model, more specific for code reviews, and test it more thoroughly than we could with a single limited experiment.

5.3 Lessons Learned

There are several problems that could have been avoided with more intensive and more realistic testing before the start of the experiment: A size limit to the size of 'post' requests to the HTTP server led to a problem with one participant in the controlled setting, a compile error introduced when seeding the defects was missed, and the time needed for participation was underestimated based on misleading results from the pre-tests. That said, the canary release strategy proved effective by avoiding further problems in the later phases of the experiment.

During experiment design, we decided to adjust the original code changes as little as possible, which meant not fixing even glaring maintenance and code style issues like empty Javadoc comments. In retrospect, this decision seems questionable. Fixing some code style issues would have threatened the experiment's realism only marginally and arguably could have even improved it. Moreover, less of these issues could have reduced noise in the data.

Another choice was to use three combinations of four different change part orders in the experiment. Comparing fewer orders or combinations would have improved statistical power for RQ_3, but on the other hand, the risk of choosing an order without an effect would have been higher, as would mono-operation bias. Therefore, we still believe the trade-off to be justified.

Others aspects worked quite well: Having an online experiment and advertising it to personal networks and interest groups allowed a quite high number of participants that would probably have been even higher with a less demanding task. By gathering demographic data on the participants and by adding various checks, *e.g.*, to counter duplicate participation, we could control many of the corresponding threats. On the down-side, the online setting probably aided the differential attrition which greatly impeded the statistical analysis for RQ_3.

6 Summary

We performed an experiment, using a browser-based experiment platform we devised, to test several hypotheses relating to the influence of cognitive load on change-based code review performance: Is the reviewer's working memory capacity associated with code review effectiveness? Does code change size/complexity influence code review effectiveness? Does the order of presenting the code change parts influence code review efficiency? We gathered usable data from 50 participants (resp. 45 for working memory), most of them professional developers, who performed (in a median time of 84 minutes) one small review and two large reviews each, with different orders of presenting the code for the large reviews. We found that working memory capacity is correlated with the effectiveness of finding delocalized defects, but not with other types of defects. We confirmed that larger, more complex code changes are associated with lower code review effectiveness. There seems to be an effect of change part ordering, too, but it is not statistically significant for our sample. Besides these main findings, our study resulted in a rich dataset of code review interaction logs that we make available together with our replication package (Baum et al, 2018).

Acknowledgements We thank all participants and all pre-testers for the time and effort they donated. We furthermore thank Sylvie Gasnier and Günter Faber for advice on the statistical procedures and Javad Ghofrani for help with double-checking the defect coding. We thank Bettina von Helversen from the psychology department at the University of Zurich for advice on the parts related to the theory of cognitive load. Bacchelli gratefully acknowledges the support of the Swiss National Science Foundation through the SNF Project No. PP00P2_170529. This is a pre-print of an article published in Empirical Software Engineering. The final authenticated version is available online at: http://dx.doi.org/10.1007/s10664-018-9676-8

References

Abdelnabi Z, Cantone G, Ciolkowski M, Rombach D (2004) 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, IEEE, pp 239–248

Agresti A (2007) An introduction to categorical data analysis, 2nd edn. Wiley
Agresti A (2010) Analysis of Ordinal Categorical Data, 2nd edn. Wiley
Bacchelli A, Bird C (2013) Expectations, outcomes, and challenges of modern code review. In: Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, pp 712–721
Balachandran V (2013) 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, IEEE Press
Barnett M, Bird C, Brunet J, Lahiri SK (2015) Helping developers help themselves: Automatic decomposition of code review changesets. In: Proceedings of the 2015 International Conference on Software Engineering. IEEE Press
Basili V, Caldiera G, Lanubile F, Shull F (1996) Studies on reading techniques. In: Proc. of the Twenty-First Annual Software Engineering Workshop, vol 96, p 002
Bates D, Maechler M, Bolker B, Walker S, et al (2014) lme4: Linear mixed-effects models using eigen and s4. R package version 1(7):1–23
Baum T, Schneider K (2016) 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, Springer, pp 301–308
Baum T, Liskin O, Niklas K, Schneider K (2016a) A faceted classification scheme for change-based industrial code review processes. In: Software Quality, Reliability and Security (QRS), 2016 IEEE International Conference on, IEEE, Vienna, Austria, DOI 10.1109/QRS.2016.19
Baum T, Liskin O, Niklas K, Schneider K (2016b) Factors influencing code review processes in industry. In: Proceedings of the ACM SIGSOFT 24th International Symposium on the Foundations of Software Engineering, ACM, Seattle, WA, USA, DOI 10.1145/2950290.2950323
Baum T, Leßmann H, Schneider K (2017a) The choice of code review process: A survey on the state of the practice. In: Product-Focused Software Process Improvement: 18th International Conference, PROFES 2017, Innsbruck, Austria, November 29 - December 01, 2017, Proceedings, Springer
Baum T, Schneider K, Bacchelli A (2017b) On the optimal order of reading source code changes for review. In: 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME), Proceedings
Baum T, Schneider K, Bacchelli A (2017c) Online material for "On the optimal order of reading source code changes for review". DOI 10.6084/m9.figshare.5236150, URL http://dx.doi.org/10.6084/m9.figshare.5236150
Baum T, Schneider K, Bacchelli A (2018) Online material for this paper, private link for review, to be published at figshare. DOI 10.6084/m9.figshare.5808609, URL https://figshare.com/s/776c11a8571501c58e7a
Bergersen GR, Gustafsson JE (2011) Programming skill, knowledge, and working memory among professional software developers from an investment the-

ory perspective. Journal of Individual Differences
Bernhart M, Grechenig T (2013) On the understanding of programs with continuous code reviews. In: Program Comprehension (ICPC), 2013 IEEE 21st International Conference on, IEEE, San Francisco, CA, USA, pp 192–198
Biegel B, Beck F, Hornig W, Diehl S (2012) The order of things: How developers sort fields and methods. In: Software Maintenance (ICSM), 2012 28th IEEE International Conference on, IEEE, pp 88–97
Biffl S (2000) 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, IEEE, pp 136–145
Chen F, Zhou J, Wang Y, Yu K, Arshad SZ, Khawaji A, Conway D (2016) Robust Multimodal Cognitive Load Measurement. Springer
Cowan N (2010) The magical mystery four: How is working memory capacity limited, and why? Current directions in psychological science 19(1):51–57
Crk I, Kluthe T, Stefik A (2016) Understanding programming expertise: an empirical study of phasic brain wave changes. ACM Transactions on Computer-Human Interaction (TOCHI) 23(1):2
Daneman M, Carpenter PA (1980) Individual differences in working memory and reading. Journal of verbal learning and verbal behavior 19(4):450–466
Daneman M, Merikle PM (1996) Working memory and language comprehension: A meta-analysis. Psychonomic bulletin & review 3(4):422–433
Denger C, Ciolkowski M, Lanubile F (2004) Investigating the active guidance factor in reading techniques for defect detection. In: Empirical Software Engineering, 2004. Proceedings. International Symposium on, IEEE, pp 219–228
DeStefano D, LeFevre JA (2007) Cognitive load in hypertext reading: A review. Computers in human behavior 23(3):1616–1641
Dias M, Bacchelli A, Gousios G, Cassou D, Ducasse S (2015) Untangling fine-grained code changes. In: Software Analysis, Evolution and Reengineering, 2015 IEEE 22nd International Conference on, IEEE, pp 341–350
Dowell J, Long J (1998) Target paper: conception of the cognitive engineering design problem. Ergonomics 41(2):126–139
Dunsmore A, Roper M, Wood M (2000) Object-oriented inspection in the face of delocalisation. In: Proceedings of the 22nd International Conference on Software Engineering, ACM, pp 467–476
Dunsmore A, Roper M, Wood M (2001) Systematic object-oriented inspection – an empirical study. In: Proceedings of the 23rd International Conference on Software Engineering, IEEE Computer Society, pp 135–144
Dunsmore A, Roper M, Wood M (2003) The development and evaluation of three diverse techniques for object-oriented code inspection. Software Engineering, IEEE Transactions on 29(8):677–686, DOI 10.1109/TSE.2003.1223643
Ebert F, Castor F, Novielli N, Serebrenik A (2017) Confusion detection in code reviews. In: 33rd International Conference on Software Maintenance and Evolution (ICSME), Proceedings, ICSME

Fagan ME (1976) Design and code inspections to reduce errors in program development. IBM Systems Journal 15(3):182–211

Falessi D, Juristo N, Wohlin C, Turhan B, Münch J, Jedlitschka A, Oivo M (2017) Empirical software engineering experts on the use of students and professionals in experiments. Empirical Software Engineering pp 1–38

Field A, Hole G (2002) How to design and report experiments. Sage

Fritz T, Begel A, Müller SC, Yigit-Elliott S, Züger M (2014) Using psycho-physiological measures to assess task difficulty in software development. In: Proceedings of the 36th International Conference on Software Engineering, ACM, pp 402–413

Geffen Y, Maoz S (2016) On method ordering. In: Program Comprehension (ICPC), 2016 IEEE 24th International Conference on

Gilb T, Graham D (1993) Software Inspection. Addison-Wesley

Gousios G, Pinzger M, Deursen Av (2014) An exploratory study of the pull-based software development model. In: Proceedings of the 36th International Conference on Software Engineering, ACM, Hyderabad, India, pp 345–355

Hassan AE (2009) Predicting faults using the complexity of code changes. In: Proceedings of the 31st International Conference on Software Engineering, IEEE Computer Society, pp 78–88

Herzig K, Zeller A (2013) The impact of tangled code changes. In: Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on, IEEE, pp 121–130

Hungerford BC, Hevner AR, Collins RW (2004) Reviewing software diagrams: A cognitive study. IEEE Transactions on Software Engineering 30(2):82–96

IEEE 24765 (2010) Systems and software engineering vocabulary iso/iec/ieee 24765: 2010. Standard 24765, ISO/IEC/IEEE

Jaccard P (1912) The distribution of the flora in the alpine zone. New phytologist 11(2):37–50

Kalyan A, Chiam M, Sun J, Manoharan S (2016) A collaborative code review platform for github. In: Engineering of Complex Computer Systems (ICECCS), 2016 21st International Conference on, IEEE, pp 191–196

Laguilles JS, Williams EA, Saunders DB (2011) Can lottery incentives boost web survey response rates? findings from four experiments. Research in Higher Education 52(5):537–553

Laitenberger O (2000) Cost-effective detection of software defects through perspective-based inspections. PhD thesis, Universität Kaiserslautern

MacLeod L, Greiler M, Storey MA, Bird C, Czerwonka J (2017) Code reviewing in the trenches: Understanding challenges and best practices. IEEE Software

Mantyla MV, Lassenius C (2009) What types of defects are really discovered in code reviews? Software Engineering, IEEE Transactions on 35(3):430–448

Matsuda J, Hayashi S, Saeki M (2015) Hierarchical categorization of edit operations for separately committing large refactoring results. In: Proceedings of the 14th International Workshop on Principles of Software Evolution, ACM, pp 19–27

McIntosh S, Kamei Y, Adams B, Hassan AE (2015) An empirical study of the impact of modern code review practices on software quality. Empirical Software Engineering pp 1–44

McMeekin DA, von Konsky BR, Chang E, Cooper DJ (2009) Evaluating software inspection cognition levels using bloom's taxonomy. In: Software Engineering Education and Training, 2009. CSEET'09. 22nd Conference on, IEEE, pp 232–239

Miller GA (1956) The magical number seven, plus or minus two: some limits on our capacity for processing information. Psychological review 63(2):81

Oswald FL, McAbee ST, Redick TS, Hambrick DZ (2015) The development of a short domain-general measure of working memory capacity. Behavior research methods 47(4):1343–1355

Paas FG, Van Merriënboer JJ (1994) Instructional control of cognitive load in the training of complex cognitive tasks. Educational psychology review 6(4):351–371

Parnas DL (1972) On the criteria to be used in decomposing systems into modules. Commun ACM DOI 10.1145/361598.361623

Pearl J (2001) Causality: models, reasoning, and inference. Cambridge University Press

Perneger TV (1998) What's wrong with bonferroni adjustments. BMJ: British Medical Journal 316(7139):1236

Platz S, Taeumel M, Steinert B, Hirschfeld R, Masuhara H (2016) Unravel programming sessions with thresher: Identifying coherent and complete sets of fine-granular source code changes. In: Proceedings of the 32nd JSSST Annual Conference

Pollock L, Vijay-Shanker K, Hill E, Sridhara G, Shepherd D (2009) Natural language-based software analyses and tools for software maintenance. In: Software Engineering, Springer, pp 94–125

Porter A, Siy H, Mockus A, Votta L (1998) Understanding the sources of variation in software inspections. ACM Transactions on Software Engineering and Methodology (TOSEM) 7(1):41–79

Rasmussen J (1983) Skills, rules, and knowledge; signals, signs, and symbols, and other distinctions in human performance models. IEEE transactions on systems, man, and cybernetics (3):257–266

Raz T, Yaung AT (1997) Factors affecting design inspection effectiveness in software development. Information and Software Technology 39(4):297–305

Rigby PC, Bird C (2013) Convergent contemporary software peer review practices. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ACM, Saint Petersburg, Russia, pp 202–212

Rigby PC, Storey MA (2011) Understanding broadcast based peer review on open source software projects. In: Proceedings of the 33rd International Conference on Software Engineering, ACM, pp 541–550

Rigby PC, Cleary B, Painchaud F, Storey M, German DM (2012) Contemporary peer review in action: Lessons from open source development. Software, IEEE 29(6):56–61

Rigby PC, German DM, Cowen L, Storey MA (2014) Peer review on open source software projects: Parameters, statistical models, and theory. ACM Transactions on Software Engineering and Methodology p 34

Robbins B, Carver J (2009) 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, IEEE Computer Society, pp 145–155

Rothlisberger D, Harry M, Binder W, Moret P, Ansaloni D, Villazon A, Nierstrasz O (2012) Exploiting dynamic information in ides improves speed and correctness of software maintenance tasks. Software Engineering, IEEE Transactions on 38(3):579–591

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

Siegmund J, Peitek N, Parnin C, Apel S, Hofmeister J, Kästner C, Begel A, Bethmann A, Brechmann A (2017) Measuring neural efficiency of program comprehension. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ACM, pp 140–150

Simon HA (1974) How big is a chunk? Science 183(4124):482–488

Singer E, Ye C (2013) The use and effects of incentives in surveys. The ANNALS of the American Academy of Political and Social Science 645(1):112–141

Sjøberg DI, Hannay JE, Hansen O, Kampenes VB, Karahasanovic A, Liborg NK, Rekdal AC (2005) A survey of controlled experiments in software engineering. Software Engineering, IEEE Transactions on 31(9):733–753

Skoglund M, Kjellgren V (2004) 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), IET

Sweller J (1988) Cognitive load during problem solving: Effects on learning. Cognitive science 12(2):257–285

Tao Y, Kim S (2015) Partitioning composite code changes to facilitate code review. In: Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on, IEEE, pp 180–190

Thongtanunam P, McIntosh S, Hassan AE, Iida H (2015a) 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

Thongtanunam P, Tantithamthavorn C, Kula RG, Yoshida N, Iida H, Matsumoto Ki (2015b) 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

Unsworth N, Heitz RP, Schrock JC, Engle RW (2005) An automated version of the operation span task. Behavior research methods 37(3):498–505

Venables WN, Ripley BD (2002) Modern Applied Statistics with S, 4th edn. Springer, New York, URL http://www.stats.ox.ac.uk/pub/MASS4, iSBN 0-387-95457-0

Walenstein A (2002) Theory-based analysis of cognitive support in software comprehension tools. In: Program Comprehension, 2002. Proceedings. 10th International Workshop on, IEEE, pp 75–84

Walenstein A (2003) Observing and measuring cognitive support: Steps toward systematic tool evaluation and engineering. In: Program Comprehension, 2003. 11th IEEE International Workshop on, IEEE, pp 185–194

Wilhelm O, Hildebrandt A, Oberauer K (2013) What is working memory capacity, and how can we measure it? Frontiers in psychology 4

Zhang T, Song M, Pinedo J, Kim M (2015) Interactive code review for systematic changes. In: Proceedings of 37th IEEE/ACM International Conference on Software Engineering. IEEE