Primers or Reminders? The Effects of Existing Review Comments on Code Review

Davide Spadini
d.spadini@sig.eu
Software Improvement Group &
Delft University of Technology
Amsterdam & Delft, The Netherlands

Gül Çalikli
gul.calikli@gu.se
Chalmers & University of Gothenburg
Gothenburg, Sweden

Alberto Bacchelli
bacchelli@ifi.uzh.ch
University of Zurich
Zurich, Switzerland

ABSTRACT

In contemporary code review, the comments put by reviewers on a specific code change are immediately visible to the other reviewers involved. Could this visibility prime new reviewers' attention (due to the human's proneness to availability bias), thus biasing the code review outcome? In this study, we investigate this topic by conducting a controlled experiment with 85 developers who perform a code review and a psychological experiment. With the psychological experiment, we find that ≈70% of participants are prone to availability bias. However, when it comes to the code review, our experiment results show that participants are primed only when the existing code review comment is about a type of bug that is not normally considered; when this comment is visible, participants are more likely to find another occurrence of this type of bug. Moreover, this priming effect does not influence reviewers' likelihood of detecting other types of bugs. Our findings suggest that the current code review practice is effective because existing review comments about bugs in code changes are *not negative primers*, rather *positive reminders* for bugs that would otherwise be overlooked during code review. Data and materials: https://doi.org/10.5281/zenodo.3653856

CCS CONCEPTS

• **Software and its engineering** → *Software verification and validation.*

KEYWORDS

Code Review, Priming, Availability Heuristic

ACM Reference Format:
Davide Spadini, Gül Çalikli, and Alberto Bacchelli. 2020. Primers or Reminders? The Effects of Existing Review Comments on Code Review. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380385

1 INTRODUCTION

Peer code review is a well-established practice that aims at maintaining and promoting source code quality, as well as sustaining development teams by means of improved knowledge transfer, awareness, and solutions to problems [3, 5, 27, 41].

In the code review type that is most common nowadays [7], the *author* of a code change sends the change for review to peer developers (also knowns as *reviewers*), before the change can be integrated in production. Previous research on three popular open-source software projects has found that three to five reviewers are involved in each review [44]. Using a software review tool, the reviewers and the author conduct an asynchronous online discussion to collectively judge whether the proposed code change is of sufficiently high quality and adheres to the guidelines of the project. In widespread code review tools, reviewers' comments are immediately visible as they are written by their authors; could this visibility bias the other reviewers' judgment?

If we consider the peer review setting for scientific articles, reviewers normally judge (at least initially) the merit of the submitted work independently from each other. The rationale behind such preference is to mitigate group members' influences on each other that might lead to errors in the individual judgments [34]. It is reasonable to think that also in code review, the visibility of existing review comments made by other developers may affect one's individual judgment, leading to an erroneous judgment.

An existing comment may prime new reviewers on a specific type of bug, due to the *availability bias* [30]. Availability bias is the tendency to be influenced by information that can be easily retrieved from memory (*i.e.*, easy to recall) [21]. This bias is one of the many cognitive biases identified in psychology, sociology, and management research [30]. Cognitive biases are systematic deviations from optimal reasoning [30, 47, 48]. In the cognitive psychology literature, Kahneman and Tversky showed that humans are prone to availability bias [51]. For example one may avoid traveling by plane after having seen recent plane accidents on the news, or may see conspiracies everywhere as a result of watching too many spy movies [21]. Therefore, it seems fitting to imagine that a reviewer may be biased toward a certain bug type, by readily seeing another reviewer's comment on such a bug type. This bias would likely result in a distorted code review outcome.

In this paper, we present a controlled experiment we devised and conducted to test the current code review setup and reviewers' proneness to availability bias. More specifically, we examine whether priming a reviewer on a bug type (achieved by showing an existing review comment) biases the outcome of code review.

Our experiment was completed by 85 developers, 73% of which reported to have at least three years of professional development experience. We required each developer to conduct a code review in which an existing comment was either shown (treatment group) or not (control group). We then measured to what extent the reviewers could find—in the same code change—(1) another bug of the same type as the primed one and (2) a bug of a different type. We created a setup with two different code changes to review.

Based on the availability bias literature, we expected the primed participants (treatment group) to be more likely to find the bug of the same type (as it is already available in memory), but less likely to find the other bug type (since distracted by the comment). Surprisingly, instead, our results show that—for three out of four bugs—the code review outcome *does not change* between the treatment and control groups. After testing our results for robustness, we could find no evidence indicating that, for these three bugs, the outcome of the review is biased in the presence of an existing review comment priming them on a bug type. Only for one bug type, though, we have strong evidence that the behavior of the reviewers changed: When the previous review comment was about a type of bug that is normally *not* considered during developers' coding/review practices (*i.e.*, checking for `NullPointerException` on a method's parameters), the reviewers were more likely to find the same type of bug with a strong effect.

Overall, we interpret the results of our experiment as an indication that existing review comments do not act as negative *primers*, rather as positive *reminders*. As such, our experiment provides evidence that the current collaborative code review practice, adopted by most software projects, could be more beneficial than separate individual reviews, not only in terms of efficiency and social advantages, but also in terms of its effectiveness in finding bugs.

2 BACKGROUND AND RELATED WORK

In this section, we review the literature on human aspects in contemporary code review practices, as well as studies on scientific peer review. Subsequently, we provide background on cognitive biases in general and present relevant studies in Software Engineering (SE). We also provide a separate subsection on availability bias, which consists of some theoretical background and existing research on availability bias in SE.

2.1 Human aspects in modern code review

Past research has provided evidence that human factors determine code review performance to a significant degree and that code review is a collaborative process [3]. Empirical studies conducted at companies such as Google [41] and Microsoft [3] revealed that, besides finding defects and ensuring maintainability, motivations for reviewing code are knowledge transfer (*e.g.*, education of junior developers) and improving shared code ownership, which is closely related to team awareness and transparency.

Besides being a collaborative activity, code review is also demanding from a cognitive point of view for the individual reviewer. A large amount of research is focused on improving code review tools and processes based on the assumption that reducing reviewers' cognitive load improves their code review performance [7, 50]. For instance, Baum *et al.* [9] argue that the reviewer and review tool can be regarded as a joint cognitive system, also emphasizing the importance of off-loading cognitive process from the reviewer to the tool. Ebert *et al.* [16] conducted a study to understand the factors that confuse code reviewers through manual analysis of 800 comments from code review of the Android project, and later they built a series of automatic classifiers (*e.g.*, Multinomial Naive Bayes, OneR) for identification of confusion in review comments. Baum *et al.* [8] conducted experiments to examine the association of working memory capacity and cognitive load with code review performance. They found that working memory capacity is associated with the effectiveness of finding de-localized defects. However, authors could not find substantial evidence on the influence of change part ordering on mental load or review performance. Spadini *et al.* [46] designed and conducted a controlled experiment to investigate whether examining changed test code before the changed production code (also known as *Test Driven Code Review* or TDR) affects code review effectiveness. According to the findings of Spadini *et al.*, developers adopting TDR find the same amount of defects in production code, but more defects in test code and fewer maintainability issues in the production code.

Significantly related to the work we present in this paper is the recent empirical observational study by Thongtanunam and Hassan [49]. They investigated the relationship between the evaluation decision of a reviewer and the visible information about a patch under review (*e.g.*, comments and votes by prior co-reviewers) [49]. With an observational study on tens of thousands of patches from two popular open-source software systems, Thongtanunam and Hassan found that (1) the amount of feedback and co-working frequency between reviewer and patch author are highly associated with the likelihood of the reviewer providing a positive vote and that (2) the proportion of reviewers who provided a vote consistent with prior reviewers is significantly associated with the defect-proneness of a patch (even though other factors are stronger). These results corroborate the hypothesis that there is some sort of influence generated by the visible information about the change under review on the behavior of the reviewers [49]. In the work we present in this paper, we setup a controlled setting to investigate an angle of this influence further, hoping to shed more light on the causal connection between comments' visibility and reviewers' effectiveness.

2.2 Scientific peer review

Peer review is the main form of group decision making used to allocate scientific research grants and select manuscripts for publication. Many studies demonstrated that individual psychological processes are subject to social influences [15]. Such finding also points out some issues that might arise during group decision making. Experimental results obtained by Deutsch and Gerard [15] show that when a group situation is created, normative social influences grossly increase, leading to errors in individual judgment. Based on the findings of this study, it is emphasized that group consensus succeeds only if groups encourage their members to express their own, independent judgments. Therefore, one of the procedures for peer review of scientific research grant applications is 'written individual review' [34]. With this review procedure, reviewers judge the merit of a grant application in written form, independently of one another, before the final decision maker approves or rejects an application. Written individual review can mitigate the influence of reviewers on the way to reach a collective judgment. It is also used in scientific venues to eliminate biases. There is also another form of review procedure, namely panel peer review where a common judgment is reached through mutual social exchange [34]. In panel peer review, a group of reviewers convene to jointly deliberate and judge the merit of an application before the funding decision is made. However, as also emphasized by Deutsch and Gerard [15], it is crucial to encourage individual members to express their own judgment without feeling under the pressure of normative social influences for proper functioning of group decision making.

2.3 Cognitive biases in software engineering

Cognitive biases are defined as systematic deviations from optimal reasoning [30, 47, 48]. In the past six decades, hundreds of empirical studies have been conducted showing the existence of various cognitive biases in humans' thought processes [21, 48]. Although many theories explain why cognitive biases exist, Baron [6] stated that there is no evidence so far about the existence of a single reason or generative mechanism that can explain the existence of all cognitive bias types. Some theories see cognitive bias as the by-product of cognitive heuristics that humans developed due to their cognitive limitations (e.g., information processing power) and time pressure, whereas some relate them to emotions.

Human cognition is a crucial part of software engineering research since software is developed by people for people. In their systematic mapping study [30], Mohanini *et al.* report 37 different cognitive biases that have been investigated by software engineering studies so far. According to the results of this systematic mapping study, the cognitive biases that are most common in software engineering studies are *anchoring bias*, *confirmation bias*, and *overconfidence bias.* Anchoring bias results from forming initial estimates about a problem under uncertainty and focusing on these initial estimates without making sufficient modifications in the light of more recently acquired information [21, 47]. Anchoring bias has so far been studied in software engineering research within the scope of requirements elicitation [37], pair programming [19], software reuse [35], software project management [2], and effort estimation [25]. Confirmation bias is the tendency to search for, interpret, favor, and recall information in a way that affirms one's prior beliefs or hypotheses [38]. The manifestations of confirmation bias during unit testing and how it affects software defect density have been widely studied in software engineering literature [11, 12, 24].

Any positive effect of experience on mitigation of confirmation bias has not been discovered so far [10]. However, in some studies, participants who have been trained in logical reasoning and hypothesis testing skills were manifested less tendency towards confirmatory behavior during software testing [10]. Ko and Myers identify confirmation bias among the cognitive biases that cause errors in programming systems [23]. Van Vliet and Tang indicate that during software architecture design, some organizations assign devil's advocate so that one's proposal is not followed without any questioning [52]. Overconfidence bias manifests when a person's subjective confidence in their judgement is reliably greater than the objective accuracy of such a judgement [31]. This bias type has been studied within the context of pair programming [19], requirements elicitation [13] and project cost estimation [26].

Availability bias. Availability bias is the tendency to be influenced by information that can be easily retrieved from memory (i.e., easy to recall) [21]. The definition of availability bias was first formulated by Tversky and Kahneman [51], who conducted a series of experiments to explore this judgemental bias. However, including these original experiments, many psychology experiments do not go beyond comparing two groups (i.e., controlled and test group) to differ in availability. To the best of our knowledge, in cognitive psychology literature, the only experiment providing evidence for the mediating process that manifests availability bias was devised by Gabrelcik and Fazio, who employed (memory) priming as the mediating process [18].

Availability bias has also been studied in SE research. De Graaf et al. [14] examined software professionals' strategies to search for documentation by using think-aloud protocols. Authors claim that using incorrect or incomplete set of keywords, or ignoring certain locations while looking for documents due to availability bias might lead to huge losses. Mohan and Jain [29] claim that while performing changes in design artifacts, developers—due to availability bias—might focus on their past experiences, since such info can be easily retrieved from developers' memory. However, such information might be inconsistent with the current state of the software system. Mohan et al. [29] propose traceability among design artifacts as a solution to mitigate the negative effects of the availability bias and other cognitive biases (i.e., anchoring and confirmation bias). Robins and Redmiles [39] propose a software architecture design environment reporting that it supports designers by addressing their cognitive challenges, including availability bias. Jørgensen and Sjøberg [20] argue that while learning from software development experience, learning from the right experiences might be hindered due to availability bias. Authors suggest retaining post-mortem project reviews to mitigate negative effects of availability bias.

Overall, existing literature points to the potential risks associated with availability bias in SE. As our community has provided evidence that code review is a collaborative and cognitively demanding process and that the collaborative nature of code review also has the potential to affect individual reviewers' cognition, availability bias could manifest itself during the code review process. This bias could hamper code review effectiveness. In our study, we aim to explore how existing review comments bias the code review outcome.

3 EXPERIMENTAL DESIGN

In this section, we explain the design of our experiment.

3.1 Research Questions and Hypotheses

The paper is structured along two research questions. By answering these research questions, we aim to understand to what extent contemporary code review is robust to reviewers' availability bias, depending on the nature of the bug for which a previous comment exists on the code change. Our first research question and the corresponding hypotheses follow.

RQ_1. *What is the effect of priming the reviewer with a bug type that is not normally considered?*

We hypothesize that an existing review comment about a bug type that reviewers do not usually consider (such as a `null` value passed as an argument [4, 7, 40, 42]) might prime the reviewers towards this bug type, so they find more of these bugs. Also, we hypothesize that—due to such priming—reviewers overlook bugs on which they were not primed. Hence, our formal hypotheses are:

$H0_{10}$**:** Priming subjects with bugs they usually do **not** consider does not affect their performance in finding bugs of the same type.

$H0_{11}$**:** Priming subjects with bugs they usually do **not** consider does not affect their performance in finding bugs they usually look for.

We also explore how priming on a bug that is usually considered during code reviews affects review performance. Therefore, our second research question is:

RQ$_2$. *What is the effect of priming the reviewer with a bug type that is normally looked for?*

We hypothesize that also in the case of an existing review comment about a bug type that reviewers usually consider primes the reviewers towards this bug type, so that they find more of these bugs. Also, we expect primed reviewers to only look for the type of bugs on which they are primed, overlooking others. Hence, our formal hypotheses are:

$H0_{20}$**:** Priming subjects with bugs they usually consider does not affect their performance in finding bugs of the same type.

$H0_{21}$**:** Priming subjects with bugs they usually consider does not affect their performance in finding bugs they usually do **not** look for.

3.2 Experiment Design and Structure

To conduct the code review experiment and to assess participants' proneness to availability bias, we extend the browser-based tool CRExperiment [43]. The tool allows us to (i) visualize and perform a code review, (ii) collect data through questions asking for subjects' demographics information as well as data consisting of participants' interactions with the tool, (iii) collect data to measure subjects' proneness to availability bias, by using a memory priming set-up to trigger subjects' use of availability heuristic that is followed by a survey. Both the priming set-up and the survey are inherited from a classic experiment in cognitive psychology literature that was designed by Gabrielcik and Fazio [18].

Code Review Experiment Overview. For the code review experiment, we follow independent measures design [22] augmented with some additional phases. The following stages in the browser-based tool correspond to the code review experiment:

(1) **Welcome Page:** The welcome page provides participants with information about the experiment. This page also aims to avoid *demand characteristics* [33], which are cues and hints that can make the participants aware of the goals of this research study leading to change in their behaviour during the experiment. For this purpose, we do not inform the participants about the full purpose of the experiment, rather they are only told that the experiment aims to compare code review performance under different circumstances. Before starting the experiment, the subjects are also asked for their informed consent.

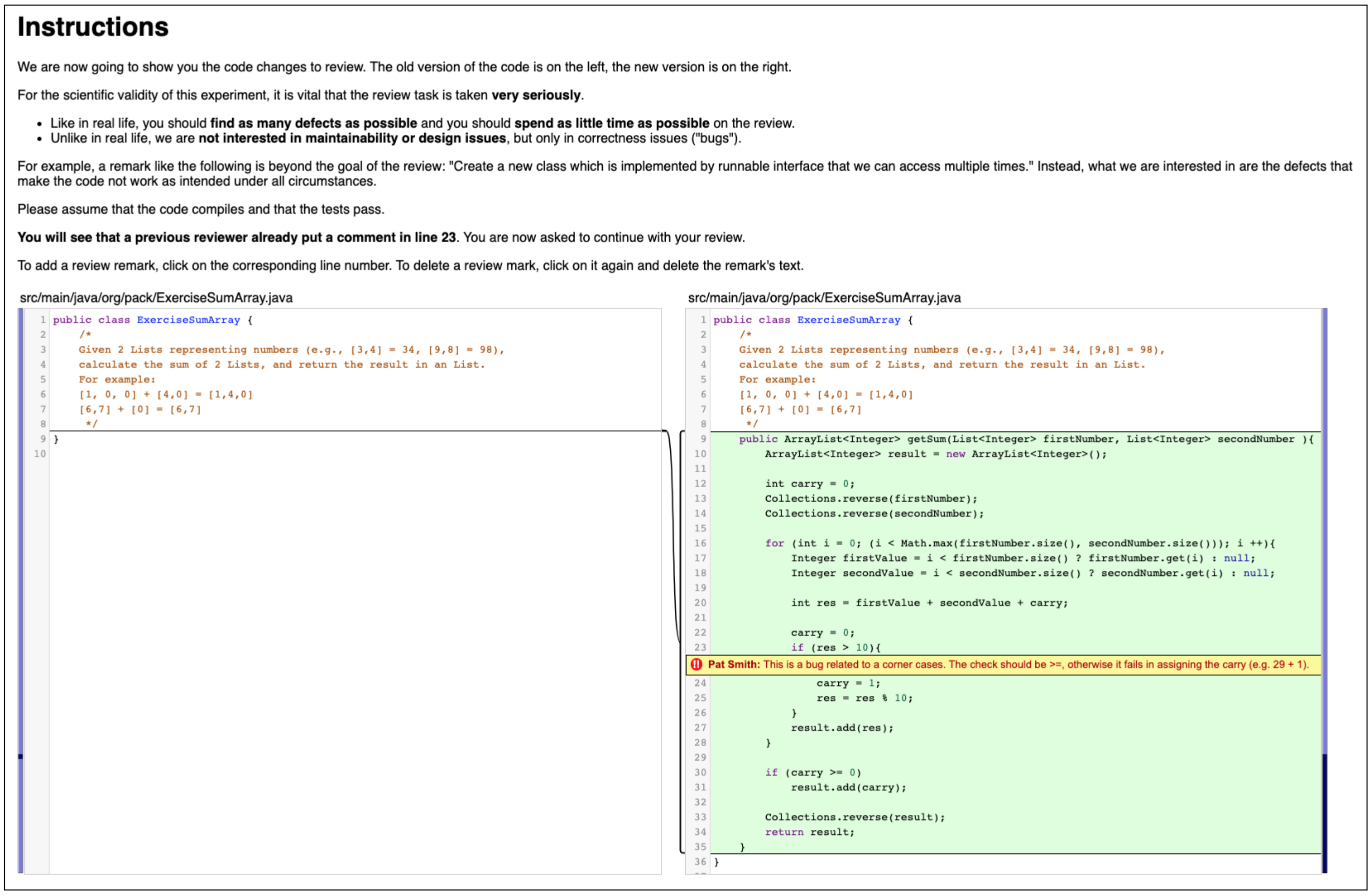
(2) **Participants' Demographics:** On the next page, subjects are asked questions to collect demographic information as well as confounding factors, such as: (i) gender, (ii) age, (iii) proficiency in the English language, (iv) highest obtained education degree, (v) main role, (vi) years of experience in software development, (vii) current frequency in software development, (viii) years of experience in Java programming, (ix) years of experience in doing code reviews, (x) current frequency of doing code reviews, and (xi) the number of hours subjects worked that day. It is kept mandatory that subjects answer these questions before proceeding to the next page where they will receive more information about the code review experiment they are about to take part in. We ask these questions to measure subjects' real, relevant, and recent experience. Collecting such data helps us to identify which portion of the developer population is represented by subjects who take part in our experiment [17].

(3) **Actual Experiment:** Each participant is then asked to perform a code review and is randomly assigned to one of the following two treatments:
   - **Pr** (*primed*)– The subject is given a code change to review where there exists a review comment (made by a previous reviewer) about a bug in the code. The test group of our experiment comprises the subjects who are assigned to this treatment.
   - **NPr** (*not–primed*)– The subject is given a code change to review. In the code change, there are no comments made by any other reviewers. The control group of our experiment comprises the subjects who are assigned to this treatment.

   More specifically, the patch to review contains three bugs: two of the same type (*i.e.*, Bug$_A$) and one of a different type (*i.e.*, Bug$_B$). In the **Pr** group, the review starts with a comment made by another reviewer showing that one instance of Bug$_A$ is present. The participant is then asked to continue the review. In the **NPr** group, the review starts without comments. The comments shown to the participants in the **Pr** group were written by the authors, and the wording was refined with the feedback from the pilots (Section 3.5). Each participant is asked to take the task very seriously. More specifically, we ask them to find as many defects as possible and, like in real life, spend as little time as possible on the review. However, unlike in real life, we ask them not to pay attention to maintainability or design issues, but only in correctness issues ("bugs"). For example, we discard comments regarding variable namings or small refactorings.

(4) **Interruptions during the Experiment:** Immediately after completing the code review, the participants are asked whether they were interrupted during the task and for how long.

(5) **Follow-up Questions:** In the last page of the code review experiment, the participants are shown the code change they just reviewed together with the bugs disclosed: For each bug, we show it and explain why it is a defect and in what cases



Instructions

We are now going to show you the code changes to review. The old version of the code is on the left, the new version is on the right.

For the scientific validity of this experiment, it is vital that the review task is taken **very seriously**.

- Like in real life, you should **find as many defects as possible** and you should **spend as little time as possible** on the review.
- Unlike in real life, we are **not interested in maintainability or design issues**, but only in correctness issues ("bugs").

For example, a remark like the following is beyond the goal of the review: "Create a new class which is implemented by runnable interface that we can access multiple times." Instead, what we are interested in are the defects that make the code not work as intended under all circumstances.

Please assume that the code compiles and that the tests pass.

You will see that a previous reviewer already put a comment in line 23. You are now asked to continue with your review.

To add a review remark, click on the corresponding line number. To delete a review mark, click on it again and delete the remark's text.

src/main/java/org/pack/ExerciseSumArray.java

```
 1 public class ExerciseSumArray {
 2     /*
 3     Given 2 Lists representing numbers (e.g., [3,4] = 34, [9,8] = 98),
 4     calculate the sum of 2 Lists, and return the result in an List.
 5     For example:
 6     [1, 0, 0] + [4,0] = [1,4,0]
 7     [6,7] + [0] = [6,7]
 8      */
 9 }
10
```

src/main/java/org/pack/ExerciseSumArray.java

```
 1 public class ExerciseSumArray {
 2     /*
 3     Given 2 Lists representing numbers (e.g., [3,4] = 34, [9,8] = 98),
 4     calculate the sum of 2 Lists, and return the result in an List.
 5     For example:
 6     [1, 0, 0] + [4,0] = [1,4,0]
 7     [6,7] + [0] = [6,7]
 8      */
 9     public ArrayList<Integer> getSum(List<Integer> firstNumber, List<Integer> secondNumber ){
10         ArrayList<Integer> result = new ArrayList<Integer>();
11
12         int carry = 0;
13         Collections.reverse(firstNumber);
14         Collections.reverse(secondNumber);
15
16         for (int i = 0; (i < Math.max(firstNumber.size(), secondNumber.size())); i ++){
17             Integer firstValue = i < firstNumber.size() ? firstNumber.get(i) : null;
18             Integer secondValue = i < secondNumber.size() ? secondNumber.get(i) : null;
19
20             int res = firstValue + secondValue + carry;
21
22             carry = 0;
23             if (res > 10){
```

Pat Smith: This is a bug related to a corner cases. The check should be >=, otherwise it fails in assigning the carry (e.g. 29 + 1).

```
24                 carry = 1;
25                 res = res % 10;
26             }
27             result.add(res);
28         }
29
30         if (carry >= 0)
31             result.add(carry);
32
33         Collections.reverse(result);
34         return result;
35     }
36 }
```

Figure 1: Example of a code review using the tool.

it might fail. Then, for each bug, we ask the participants to indicate whether they captured it in the review:

- If the participants found the bug and they belonged to the **Pr** group, we ask them to what extent the comment of the previous reviewer influenced the discovery of the bug (using a 5-point Likert scale).
- If the participants did not find the bug (independently whether they were in the **Pr** or **NPr** group), we ask them to elaborate on why they think they missed the bug.

Assessment of Proneness to Availability Bias. The code review experiment is followed by a set-up that primes participants' memory to trigger availability bias. This set-up serves as a mediating process to manipulate availability bias so that we can measure the extent to which each subject is prone to this type of cognitive bias. To measure this phenomenon, we inherited the test part of the controlled experiment of Gabrielcik and Fazio [18]. In the original experiment, the difference in the results of control and test groups showed that (memory) priming triggered the participants' availability biases. There are three reasons why we selected this experiment for assessing the proneness to availability bias: (i) To the best of our knowledge, it is the only experiment where the underlying cognition mechanism (*i.e.*, memory priming) that triggers availability bias is explicitly devised; (ii) memory priming mechanism is also employed in code review experiment to trigger participants' availability bias; and (iii) survey in the original experiment makes it possible to quantitatively assess participants' proneness to availability bias. Therefore, the remaining stages in the browser-based tool comprise the following:

(1) **Welcome Page:** We provide a second welcome page in which, to avoid demand characteristics [33], the participants are told that they are about to participate in an experiment that aims to explore software engineers' attention by testing a set of visual stimuli, instead of the actual goal.
(2) **Warm-up Session:** We proceed with a warm-up session in which participants are asked to focus on a series of 20 words flashing once each on the screen. The words are randomly selected from the English dictionary, and none of them contain the letter 'T'. Each word flashes for 300ms. At the end of the warm-up, we ask the participants to write three words they have seen and recall, and to make a guess if they do not remember them.
(3) **Actual Psychology Experiment:** After the warm-up, we proceed with the actual psychology experiment: this time, we show two series of 20 words, all of them including the

letter 'T'. This time words flash at a faster rate, *i.e.*, 150*ms*, to avoid that the participants consciously recognize that the words have the letter 'T' so often, which would bias their last task [18]. After each series, we ask the participant to write three words they have seen and recall, and to make a guess if they do not remember them.

(4) **Measuring Proneness to Availability Bias:**The last task of the participants is to answer 15 questions, which ask to compare the frequency words for a given pair of letters in the English dictionary. For example, given the question "Do more words contain T or S", participants responded on a 9-point scale, with one end labeled "Many more contain T" and the other "Many more contain S". Our main goal is to measure the extent to which each subject is prone to availability bias. Hence, in 5 of the 15 questions we ask whether in the English dictionary there are more words containing the letter 'T' or another random letter. As in the experiment of Gabrielcik and Fazio [18], we expect the participants to indicate that there are more words containing the letter 'T' (even though this is not the case) since they were primed in step 3. The other 10 questions are used to prevent the participants from understanding the actual aim of the study.

3.3 Objects

The objects of the study are represented by the code changes (or patch, for brevity) to review, and the bugs that we selected and injected, which must be discovered by the participants.

Patches. To avoid giving some developers an advantage, the two patches are not selected from open-source software projects, hence they are not known to any of the participants. To maintain the difficulty of the code review reasonable (after all, developers are used to review only the codebase on which they work every day), we screen many websites that offer Java exercises searching for exercises that are: (1) neither too trivial nor too complicated (based on our experience teaching programming to students), (2) self-contained, and (3) do not rely on special technologies or frameworks/libraries.

After several brain-storming sessions among the authors, only two exercises satisfied these goals and were selected.

Defects. Code review is a well-established and widely adopted practice aimed at maintaining and promoting software quality [3, 41]. There are different reasons on why developers adopt this practice, but one of the main ones is to detect defects [3]. Hence, in our experiment we manually seed bugs (functional defects) in the code. More specifically, we seed two different types of bugs: one that could cause a `NullPointerException` (Bug$_A$), and one that could cause the return of a wrong value (Bug$_B$).

The bugs were injected in the code as follows:

- In Patch$_1$, we inject two Bug$_A$ and one Bug$_B$ (the priming is done on Bug$_A$),
- In Patch$_2$, we inject two Bug$_B$ and one Bug$_A$ (the priming is done on Bug$_B$),

The `NullPointerException` (Bug$_A$) in the first change was on the passed parameters. As reported by white [7] and gray literature [4, 40, 42], developers are not used to check for this kind of errors in code review, because they expect the caller to make sure the parameters are not null: hence, we use it as the *not normally considered bug* that we investigate in RQ$_1$. Instead, Bug$_A$ in the second change (RQ$_2$) does not regard a parameter, to make sure that it is bug type that normally developers look for in a review.

3.4 Variables and Measurement Details

We aim to investigate whether participants that are primed on a specific type of bug are more likely to capture only that type of bug. To understand whether the subjects did find the bug (*i.e.*, the value for our dependent variables), we proceed with the following steps: (1) the first author of this paper manually analyzes all the remarks added by the participants (each remark is classified as identifying a bug or being outside of the study's scope), then (2) the authors cross-validate the results with the answer given by the participants (as explained in Section 3.2, after the experiment the participants had to indicate whether they captured the bugs).

In Table 1, we represent all the variables of our model. The main independent variable of our experiment is the treatment (**Pr** or **NPr**). We consider the other variables as control variables, which also include the time spent on the review, the participant's role, years of experience in Java and Code Review, and tiredness. Finally, we run a logistic regression model similar to the one used by McIntosh*et al.* [28] and Spadini *et al.* [46]. To ensure that the selected logistic regression model is appropriate for the available data, we first (1) compute the Variance Inflation Factors (VIF) as a standard test for multicollinearity, finding all the values to be below 3 (values should be below 10), thus indicating little or no multicollinearity among the independent variables, (2) run a multilevel regression model to check whether there is a significant variance among reviewers, but we found little to none, thus indicating that a single level regression model is appropriate, and, finally, (4) when building the model we added the independent variables step-by-step and found that the coefficients remained stable, thus further indicating little to no interference among the variables. For convenience, we include the script to our publicly available replication package [45].

Availability bias score. We calculate availability bias scores as in the original experiment by Gabrielcik and Fazio [18]. The frequency comparisons on the 9–point scale were scored by assignments of a value between +4 and −4. Positive numbers were assigned for ratings indicating that letter 'T' was contained in more words than the other letter, while negative numbers were assigned in favour of the other letter. We calculated the availability bias score for each participant as the average (and also median) of values for the 5 relevant questions.

3.5 Pilot Runs

As the first version of the experiment was ready, we started conducting pilot runs to (1) verify the absence of technical errors in the online platform, (2) check the ratio with which participants were able to find the injected bugs (regardless of their treatment group), (3) tune the experiment on the proneness to availability bias (in terms of flashing speed and number of words to ask), (4) verify the understandability of the instructions as well as the user interface, and (5) gather qualitative feedback from the participants. We conducted three different pilot runs, for a total of 20 developers. The participants were recruited through the professional network of the study authors to ensure that they would take the task seriously and

Table 1: Variables used in the statistical model.

Metric	Description
Dependent Variables	
FoundPrimed	The participant found the bug that was primed
FoundNotPrimed	The participant found the bug that was not primed
Independent Variable	
Treatment	Type of the treatment (**Pr** or **NPr**)
Control Variables	
Gender	Gender of the participant
EnglishLevel	English Level
Age	Age of the participant
LevelOfEducation	Highest achieved level of education
Role	Role of the participant
ProfDevExp	Years of experience as professional developer
JavaExp	Years of experience in Java
ProgramPractice	How often they program
ReviewPractice	How often they perform code review
ReviewExp	Years of experience in code review
WorkedHours	Hours the participant worked before performing the experiment
Tired	How tired was the participant at the moment of taking the experiment
Stressed	How stressed was the participant at the moment of taking the experiment
Interruptions	For how long the participant was interrupted during the experiment
TotalDuration	Total duration of the experiment
PsychoExpIsPrimed	Whether the participant was primed in the psychology experiment

(†) see Figure 2 for the scale

provide feedback on their experience. No data gathered from the 20 participants to the pilot was considered in the final experiment.

After each pilot run, we inspected the results and the qualitative feedback we received and discussed extensively among the authors to verify whether parts of the experiment should have been changed. After the third run, the required changes were minimal, and we considered the experiment ready for its main run.

3.6 Recruiting Participants

The experiment was spread out through practitioners blogs and web forums (*e.g.*, Reddit) and through direct contacts from the professional network of the study authors, as well as the authors' social media accounts on Twitter and Facebook. We did not reveal the aim of the experiment. To provide a small incentive to participate, we introduced a donation-based incentive of five USD to a charity per valid respondent.

4 THREATS TO VALIDITY

Construct Validity. Threats to construct validity concern our research instruments. To measure the extent to which subjects are prone to availability bias, we used the memory priming mechanism and the survey that was employed in an experiment designed and conducted by Gabrielcik and Fazio, in cognitive psychology literature [18]. Data obtained from the controlled experiment that Gabrielcik and Fazio conducted provide direct evidence that memory priming can be a mediating process to trigger availability bias. The remaining constructs we use are defined in previous publications, and we reuse the existing instruments as much as possible. For instance, the tool employed for the online experiment is based on similar tools used in earlier works [9, 46].

To avoid problems with experimental materials, we employed a multi-stage process: After tests among the authors, we conducted three experiments with ≈7 subjects each time (for a total of 20 pilots) with external participants. After each pilot session, we made corrections to the experiment based on the feedback from the subjects of the pilot, materials were checked by the authors one more time before we launched the actual experiment.

Regarding defects and code changes, the first author prepared the code changes and corresponding test codes as well as injecting the defects into these code changes. These were later checked by the other authors. Code change and corresponding test code were on the same page, and subjects had to scroll down to proceed to the next page of the online experiment. In this way, we aimed to ensure that subjects saw the test code. Test code were added to make the experiment closer to a real world scenario.

A major threat is that the artificial experiment created by us could differ from a real-world scenario. We mitigated this issue by (1) re-creating as close as possible a real code change (for example, submitting test code and documentation together with the production code), and (2) using an interface that is identical to the common Code Review tool Gerrit [1] (both our tool and Gerrit use Mergely [36] to show the diff, also using the same color scheme).

Internal Validity. Threats to internal validity concern factors that might affect the cause and effect relationship that is investigated through the experiment. Due to the online nature of the experiment, we cannot ensure that our subjects conducted the experiments with the same set-up (*e.g.*, noise level and web searches), however we argue that developers in real world settings also have a multi-fold of tools and environments. Moreover, to mitigate the possible threat posed by missing control over subjects, we included some questions to characterize our sample (*e.g.*, experience, role, and education).

To prevent duplicate participation, we adjusted the settings of the online experiment platform so that each subject can take the experiment only once. To exclude participants who did not take the experiment seriously, we screened each review and we did not consider experiments without any comments in the review, that took less than five minutes to be completed, or that were not completed at all.

Furthermore, several background factors (*e.g.*, age, gender, experience, education) may have impact on the results. Hence, we collected all such information and investigated how these factors affect the results by conducting statistical tests.

External Validity. Threats to external validity concerns the generalizability of results. To have a diverse sample of subjects (representative of the overall population of software developers who employ contemporary code review), we invited developers from several countries, organizations, education levels, and background.

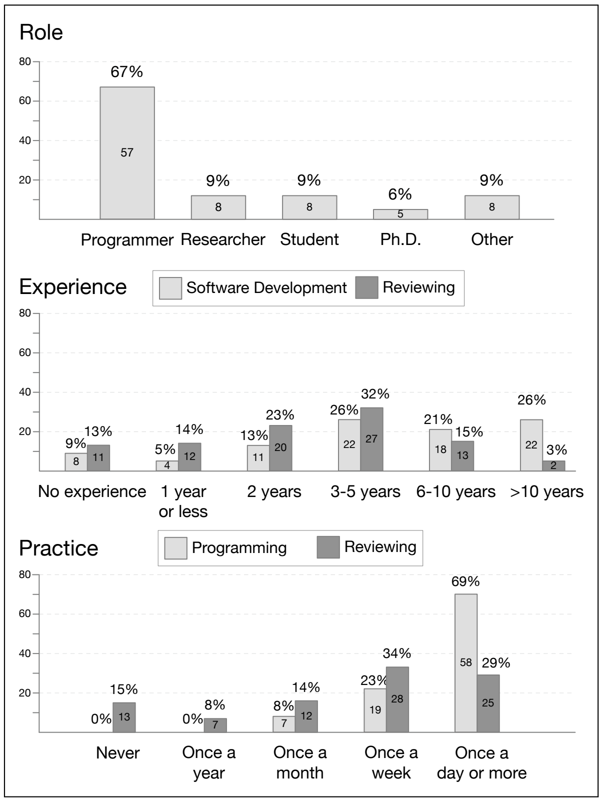


Figure 2: Participants' characteristics

5 RESULTS

In this section, we report the results of our investigation on whether and how having a comment from a previous reviewer influences the outcome of code review.

5.1 Validating The Participants

A total of 243 people accessed our experiment environment following the provided link. From these participants, we exclude all the instances in which the code change is skipped or skimmed, by demanding either at least one entered remark or more than five minutes spent on the review. After applying the exclusion criteria, a total of 85 participants are selected for the subsequent analyses.

Figure 2 presents the descriptive statistics on what the participants reported in terms of their role, experience, and practice. The majority of the participants are programmer (67%) and reported to have many years of experience in professional software development (73% more than 3 years, 47% more than 6); most program daily (69%) and review code at least weekly (63%).

Table 2 represents how the participants' are distributed across the considered treatments and code changes. The automated assignment algorithm allowed us to obtain a rather balanced number of reviews per treatment and code change.

Table 2: Distribution of participants ($N = 85$) across the various treatment groups.

	Primed (Pr)	Not Primed (NPr)	Total
CodeChange1	21	17	38
CodeChange2	22	25	47
Total	43	42	

Table 3: Odds ratio for capturing the primed and not primed bug in the test (Pr) and control (NPr) group.

Primed bug (NPE)	Primed (Pr)	Not Primed (NPr)	Total
found	13	2	15
not found	8	15	23
		Odds Ratio: 12.19 (2.19, 67.94)	
		$p < 0.001$	

Not primed bug	Primed (Pr)	Not Primed (NPr)	Total
found	14	14	28
not found	7	3	13
		Odds Ratio: 0.43 (0.09, 2.00)	
		$p = 0.275$	

5.2 RQ$_1$. Priming a not commonly reviewed bug

To investigate our first research question, the participants in our test group (**Pr**) are primed on a `NullPointerException` (NPE) bug in a method's parameter. We expect this type of bug to be missed by most not primed reviewer, because normally reviewers would assume that parameters are checked from the calling function [4, 40, 42].

Table 3 reports the results of the experiment by treatment group. From the first part of the table (primed bug), we can notice that participants in the **Pr** group found the other NPE bug 62% of the times, while participants in the **NPr** group only 11%. Expressed in odds, this result means that the NPE defect is 12 times more likely to be found by a participant in the **Pr** group. The main reasons reported by the participants in the **NPr** for missing this bug are that (1) they were too focused on the logic and not thoroughly enough when it comes the corner cases, (2) did not put attention to the fact that Integer could be null, and (3) that they generally do not check for NPE, but assume to not receive a wrong object as an input.

As expected, even though NullPointerException has been reported to be the most common bug in Java programs [53], developers stated they rarely sanity check the Object. However, as shown in Table 3, the result drastically changes when a previous reviewer points out that an NPE could be raised: in this case, many of the participants in the **Pr** group looked for other NPE bugs in the code.

When we look at whether the **Pr** group was primed by the previous reviewer comment (hence whether they were able to capture the bug because of they have been primed), we have that 40% indicated they were 'Extremely influenced', 40% were 'Very influenced' and 20% instead were 'Somewhat influenced'. Hence, the reviewers perceived to have been influenced by the existing comment.

We find a statistically significant relationship ($p < 0.001$, assessed using χ^2) of strong positive strength ($\phi = 0.5$) between the

Table 4: Regressions for primed and not primed bugs.

	Primed bug			Not primed bug		
	Estimate	S.E.	Sig.	Estimate	S.E.	Sig.
Intercept	0.704	4.734		-0.893	4.093	
IsPrimed	3.627	1.320	**	-1.199	1.073	
TotalDuration	0.001	0.002		0.003	0.001	.
ProfDevExp	0.813	0.557		-0.503	0.554	
ProgramPractice	-0.096	0.828		-0.243	0.736	
ReviewExp	-0.070	0.630		-0.813	0.651	
ReviewPractice	-1.152	0.758		1.243	0.643	.
Tired	-0.834	0.832		0.517	0.651	
WorkedHours	-0.069	0.196		0.305	0.207	
Interruptions	-1.752	0.758	*	-0.715	0.444	
... (†)						

*significance codes: '***'p <0.001, '**'p <0.01, '*'p <0.05, '.'p <0.1*
(†) Role is not significant and omitted for space reason

presence of the comment and whether the same type of bug was found. Therefore, we can reject $H0_{10}$.

Considering the second part of Table 3, we see that the not primed bug was found by both groups (**Pr** and **NPr**) at similar rate. For the former, participants found it 66% of the times, while in the **NPr** they found it 82% of the times. As shown in the table, the difference is not statistically significant ($p = 0.275$).

When looking at the participants' comments on why they missed this bug, we have that the main reasons are (1) that they forgot to try the specific corner case, and (2) that they assumed tests were covering all the corner cases. The reasons for not capturing the defects were similar in both groups. Given this result, we cannot reject $H0_{11}$. Priming the participants on a specific type of bug did **not** prevent them from capturing the other type of bug.

In Table 4 we show the result of our statistical model, taking into account the characteristics of the participants and reviews. The model confirms the result shown in Table 3: even taking into account all the variables, the **isPrimed** variable is statistically significant exclusively for the primed bug. The other variable statistically significant in the model is 'Interruptions', that is the number of times the participant has been interrupted during the experiment: the estimate has a negative value, which means the higher the number of 'Interruptions', the lower the number of bugs captured, as one can expect.

For the not primed bug instead, none of the variables are statistically significant (with 'TotalDuration' and 'ReviewExp' are slightly significant, with $p < 0.1$)

> ***Finding 1***. *Reviewers primed on a bug that is not commonly considered are more likely to find other occurences of this type of bugs. However, this does not prevent them in finding also other types of bugs.*

5.3 RQ2. Priming on an algorithmic bug

To investigate our second research question, the participants in our test group (**Pr**) are primed on an algorithmic bug, more specifically a corner case (CC) bug. The result of this experiment is shown in

Table 5: Odds ratio for capturing the primed and not primed bug in the test (Pr) and control (NPr) group.

Primed bug (CC)	Primed (Pr)	Not Primed (NPr)	Total
found	10	8	18
not found	12	17	29

Odds Ratio: 1.77 (0.54, 5.81)
$p = 0.344$

Not primed bug	Primed (Pr)	Not Primed (NPr)	Total
found	13	16	29
not found	9	9	18

Odds Ratio: 0.81 (0.25, 2.64)
$p = 0.73$

Table 5. Participants in both groups found the primed bug ~50%. Indeed, the difference is not statistically significant ($p = 0.344$). If we consider whether the test group was primed by the previous reviewer comment, 50% of the participants reported that they were 'Extremely influenced', 10% was 'Somewhat influenced' and 40% was slightly or not influenced; thus suggesting that even the reviewers noticed a lower influence from this comment, even though it was of the same type as one of the other two bugs in the same code change.

Among the main reasons for missing the bug, participants mainly stated that (1) the tests drove them to not remember that corner case, and (2) they focused more on the first one. Hence, given this result we can conclude that the participants who saw the review comment did **not** find the similar bug more often than the participants that did not see the review comment.

In the second part of Table 5, we indicate whether the participants were able to find the not primed bug. Both the test and control group are very similar in this case, too. Indeed, in both groups the bug is found around 50% of the times and the difference is not statistically significant. When looking at the participants' comments on why they missed this bug, the main reasons they state are (1) that they were too focused on capturing algorithmic bugs without paying attention to NPE, and (2) that, as in the previous RQ, they did not put attention to the fact that `Integer` could be `null`.

Given these results, we cannot reject $H0_{20}$ nor $H0_{21}$.

In Table 6, we show the result when controlling for other variables. Our dependent variable **IsPrimed** is not statistically significant. However, we see that 'TotalDuration' (*i.e.*, the time required by the developer to complete the code review) is statistically significant and in the expected direction. For the **NPr** group, the only variable that is significant is 'ReviewPractice' (*i.e.*, the average number of time the participant perform code reviews). Both these results are in line with what found in previous research [8].

> ***Finding 2***. *Reviewers primed on an algorithmic bug perceive an influence, but are as likely as the others to find algorithmic bugs. Furthermore, primed participants did not capture fewer bugs of the other type.*

Table 6: Regressions for primed and not primed bugs.

	Primed bug			Not primed bug		
	Estimate	S.E.	Sig.	Estimate	S.E.	Sig.
Intercept	-1.0510159	2.2460623		-3.037e-01	2.568e+00	
IsPrimed	0.9260383	0.7223408		-1.670e-01	7.740e-01	
TotalDuration	0.0018592	0.0008958	*	9.561e-05	9.976e-04	
ProfDevExp	-0.6031309	0.3381302	.	-9.437e-02	3.721e-01	
ProgramPractice	0.0319636	0.5905427		-1.061e+00	7.353e-01	
ReviewExp	0.3411589	0.4548836		1.284e-01	4.660e-01	
ReviewPractice	0.1531502	0.3784472		1.211e+00	4.683e-01	**
Tired	0.0835410	0.3706085		2.486e-01	4.539e-01	
WorkedHours	-0.1619234	0.1184626		2.257e-01	1.542e-01	
Interruptions	-0.1755182	0.3220796		-1.331e-01	3.630e-01	
... (†)						

*significance codes: '***' $p < 0.001$, '**' $p < 0.01$, '*' $p < 0.05$, '.' $p < 0.1$*
(†) Role is not significant and omitted for space reason

5.4 Robustness Testing

In the previous sections, we presented the results of our study on whether and to what extent reviewers can be primed during code review by showing an existing code review comment. Surprisingly, the results showed that many of our hypotheses were not satisfied: in our experiment, only in one case primed reviewers captured more bugs than the not primed group; in all the other cases, reviewers from both groups could capture the same bugs.

To further challenge the validity of these findings, in this section, we employ *robustness testing* [32]. For this purpose, we test whether the results we obtained by our baseline model hold when we systematically replace the baseline model specification with the following plausible alternatives.

Bugs were too simple or too complicated to find. Choosing the right defects to inject in the code change is fundamental to the validity of our results. If a defect is too easy to find, participants might find the bugs regardless of any other influencing factor, even without paying too much attention to the review (on the other hand, if it is too complicated reviewers might not find any bug and get discouraged to continue). We measure that ~50% of the participants found the three types of defects that we expected them to find, thus ruling out the possibility that these bugs were either too trivial or too difficult to find.

People were not primed. The entire experiment is based on the premise that reviewers in the **Pr** group were correctly primed. Even though we can not verify this premise (the experiment is online, hence there is no interaction between the researchers and the participants), after the code review experiment the participants had to indicate whether they were influenced by the comment of the previous reviewer in capturing the bug. As we stated in Section 5.2 and Section 5.3, 70% of the participants indicated they were extremely or very influenced, while only 18% indicated somewhat or slightly influenced (12% were neutral). This gives an indication that the participants felt they were indeed primed, but this did not influence their ability to find other bugs.

Nevertheless, the reported level of being influenced is subjective, so not fully reliable (participants could think to have been influenced, but were not). To triangulate this result, we test another possibility: More specifically, one of the possible explanations of why participants may not have been primed is that our sample of participants was "immune" to priming or very difficult to prime. Indeed, there is no study that confirms that developers are as affected by priming as the general population (on which past experiment was conducted). To rule out this possibility, we devised the psychological experiment: We tested whether developers can also be primed as expected using visual stimuli. Our results show that ~70% of the participants were primed as expected.

Not enough participants. Another possibility of why we do not find a difference is that we did not have enough participants. Even though 85 participants is quite large in comparison to many experiments in software engineering [8] and we tried to design an experiment that would create a strong signal, we cannot rule out that the significance was missing due to the number of participants. However, even if the results were statistically significant (assuming we had the same ratios, but an order of magnitude more of participants), the size of the effect (calculated using the ϕ coefficient) would be 'none to very negligible'. This suggests that there was no emerging trend and that, even having more participants, we could have probably obtained a significant, yet trivial effect.

Some participants did not perform the task seriously. Finally, one of the reasons why we did not confirm most of our hypotheses could be that some participants did not take the task seriously, hence they might have performed poorly and have altered the results. Having used a random assignment and having a reasonably large number of participants, we have no reason to think that one group had more 'lazy' participants than the others. Moreover, as we discussed in Section 3, to exclude participants who did not take the experiment seriously, we filtered out experiments without any comments in the review (even if there were comments, the first author manually validated them to check whether they were appropriate and they were/not capturing a bug); we also did not consider reviews that took less than five minutes to be completed, or that were not completed at all (maybe because the participant left after few minutes).

Alternatively, it would be possible that participants who were more serious focused more and found more bugs (regardless of the priming), while less serious ones would just find one and leave the experiment. To test also this possibility, we compared the likelihood of a participant in finding a second bug when a first one was found. Also in this case, we did not find any statistically significant effect, thus ruling out this hypothesis as well.

6 DISCUSSIONS

We discuss the main implications and results of our study.

Robustness of code review against availability bias. The current code review practice expects reviewers to review and comment on the code change asynchronously, and reviewers' comments are immediately visible to both the author and other reviewers.

One of the main hypotheses we stated in our study is that the code review outcome is biased because reviewers are primed by the visibility of existing comment on a bug. Indeed, if reviewers get primed by previously made comments about some bug(s), then they could find more bugs of that specific type while overlooking other types of bugs. This would, in turn, undermine the effectiveness of the code review process, creating a demand for a different approach.

To create a different approach, one might consider adopting a review method similar to that of scientific venues where reviewers do not see the comments of the other reviewers until they submit their review. Even though this strategy would reduce the transparency of the code review process undermining knowledge transfer, team awareness, as well as shared code ownership, and would probably lead to a loss in review efficiency due to duplicate bug detection, it would be necessary if the biasing effect of other reviewers' comments would be strong.

Our experiment results show that the participants in the test group were *positively* influenced by the existing comment on the code change so that they could capture more bugs of the same type. However, unexpectedly, they were still able to capture the bugs of the different type as the control group did. Like any human, reviewers are also prone to availability bias [21] to various extents. However, our results did not find evidence of a strong negative effect of reviewers' availability bias. Therefore, our data does not provide any evidence that would justify a change in the current code review practices.

Existing comments on normally not considered bugs act as (positive) *reminders* rather than (negative) *primers*. Surprisingly, participants in the test group who were primed with the algorithmic bug type (more specifically, a corner case bug) detected the same amount of corner case and `NullPointerException` (NPE) bugs as the participants in the control group. However, participants who were primed with a bug that is normally not considered in review (*i.e.*, NPE) were 12 times more likely to capture this type of bug, than the participants of the control group.

This result shows that existing reviewer comments on code change seem to support recalling (*i.e.*, act as a reminder), rather than distracting the reviewer. As previously mentioned in section 5.2, participants in the test group indicated that they were focused on to the corner cases in the code change and did not put attention to the possibility that *Integer* could be *null*. Such feedbacks are in-line with the possible existence of *anchoring bias* [21, 47].

It is likely that the existence of a reviewer comment on a uncommon bug had a de-biasing effect on the participants in the test group (*i.e.*, mitigated the participants' bias). In software engineering literature, there are empirical studies on practitioners' anchoring bias. For instance, Pitts and Brown [37] provide procedural prompts during requirements elicitation to aid analysts not anchoring on currently available information. According to the findings by Jain *et al.* [19], pair programming novices tend to anchor to their initial solutions due to their inability to identify such a wider range of solutions. However, to the best of our knowledge, there are no studies on anchoring bias within the context of code reviews. Therefore, further research is required to investigate underlying cognition mechanisms that can explain why existing reviewer comments on the unexpected bug act as *reminders*.

7 CONCLUSIONS

In the study presented in this paper, we investigated robustness of peer code review against reviewers' proneness to availability bias. For this purpose, we designed and conducted an online experiment with 85 participants, including a code review task and a psychological experiment. With the psychological experiment, the majority of the participants (i.e., ≈70%) were assessed to be prone to availability bias (median = 3.8, max = 4). However, when it comes to the code review, our experiment results show that participants are primed only when the existing code review comment is about a type of bug that is not normally considered; when this comment is visible, participants are more likely to find another occurrence of this type of bug. Hence, existing comments on this type of bugs acted as *reminders* rather than *primers*. It is our hope that this study is replicated by other researchers to gain further insights about the extent of robustness of peer code review.

ACKNOWLEDGMENTS

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No. 642954. Alberto Bacchelli gratefully acknowledges the support of the Swiss National Science Foundation through the SNF Project No. PP00P2_170529.

REFERENCES

[1] 2019. Gerrit Code Review. https://www.gerritcodereview.com.
[2] T. K. Abdel-Hamid, K. Sengupta, and D. Ronan. 1993. Software project control: an experimental investigation of judgment with fallible information. *IEEE Transactions on Software Engineering* 19, 6 (June 1993), 603–612. https://doi.org/10.1109/32.232025
[3] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*. 712–721. https://doi.org/10.1109/ICSE.2013.6606617
[4] Baeldung. 2018. Avoid Check for Null Statement in Java. https://www.baeldung.com/java-avoid-null-check.
[5] Richard A Baker Jr. 1997. Code reviews enhance software quality. In *Proceedings of the 19th International Conference on Software Engineering*. ACM, 570–571.
[6] Jonathan Baron. 2009. *Thinking and Deciding*. Cambridge University Press.
[7] Tobias Baum and Kurt Schneider. 2016. On the Need for a New Generation of Code Review Tools. In *Product-Focused Software Process Improvement*, Pekka Abrahamsson, Andreas Jedlitschka, Anh Nguyen Duc, Michael Felderer, Sousuke Amasaki, and Tommi Mikkonen (Eds.). Springer International Publishing, Cham, 301–308.
[8] Tobias Baum, Kurt Schneider, and Alberto Bacchelli. [n.d.]. Associating working memory capacity and code change ordering with code review performance. *Empirical Software Engineering* ([n. d.]), 1–37.
[9] Tobias Baum, Kurt Schneider, and Alberto Bacchelli. 2017. On the Optimal Order of Reading Source Code Changes for Review. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 329–340. https://doi.org/10.1109/ICSME.2017.28
[10] Gul Calikli and Ayse Bener. 2015. Empirical analysis of factors affecting confirmation bias levels of software engineers. *Software Quality Journal* 23, 4 (01 Dec 2015), 695–722. https://doi.org/10.1007/s11219-014-9250-6
[11] G. Calikli, A. Bener, T. Aytac, and O. Bozcan. 2013. Towards a Metric Suite Proposal to Quantify Confirmation Biases of Developers. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. 363–372. https://doi.org/10.1109/ESEM.2013.47
[12] Gül Çalıklı and Ayşe Başar Bener. 2013. Influence of confirmation biases of developers on software quality: an empirical study. *Software Quality Journal* 21, 2 (01 Jun 2013), 377–416. https://doi.org/10.1007/s11219-012-9180-0
[13] Suranjan Chakraborty, Saonee Sarker, and Suprateek Sarker. 2010. An Exploration into the Process of Requirements Elicitation: A Grounded Approach. *Journal of the Association for Information Systems* 11 (2010), 212–249.
[14] Klaas Andries de Graaf, Peng Liang, Antony Tang, and Hans van Vliet. 2014. The Impact of Prior Knowledge on Searching in Software Documentation. In *Proceedings of the 2014 ACM Symposium on Document Engineering (DocEng '14)*. ACM, New York, NY, USA, 189–198. https://doi.org/10.1145/2644866.2644878
[15] M. Deutsch and H. B. Gerard. 1955. A study of normative and informational social influences upon individual judgment. *The Journal of Abnormal and Social Psychology* 51, 3 (1955), 629–636. https://doi.org/10.1147/sj.153.0182
[16] F. Ebert, F. Castor, N. Novielli, and A. Serebrenik. 2017. Confusion Detection in Code Reviews. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 549–553. https://doi.org/10.1109/ICSME.2017.40
[17] Davide Falessi, Natalia Juristo, Claes Wohlin, Burak Turhan, Jürgen Münch, Andreas Jedlitschka, and Markku Oivo. 2018. Empirical software engineering experts on the use of students and professionals in experiments. *Empirical*

Software Engineering 23, 1 (01 Feb 2018), 452–489. https://doi.org/10.1007/s10664-017-9523-3

[18] Adele Gabrielcik and Russell H. Fazio. 1984. Priming and Frequency Estimation: A Strict Test of the Availability Heuristic. *Personality and Social Psychology Bulletin* 10, 1 (1984), 85–89. https://doi.org/10.1177/0146167284101009

[19] Radhika Jain, Jaime Muro, and Kannan Mohan. 2006. A Cognitive Perspective on Pair Programming. In *AMCIS 2006 Proceedings (AMCIS 2006)*. 444.

[20] Magne Jorgensen and D. Sjoberg. 2000. The Importance of not Learnig from Experience. In *Proceedings of European Software Process Improvement.*

[21] Daniel Kahneman. 2011. *Thinking Fast and Slow.* Farrar, Strauss, Giroux.

[22] R. E. Kirk. 2013. *Experimental Design: Procedures for the Behavioral Sciences.* SAGE Publications.

[23] Andrew Jensen Ko and Brad A. Myers. 2005. A framework and methodology for studying the causes of software errors in programming systems. *J. Vis. Lang. Comput.* 16 (2005), 41–84.

[24] Laura M. Leventhal, Barbee Teasley, Diane S. Rohlman, and Keith Instone. 1993. Positive Test Bias in Software Testing Among Professionals: A Review. In *Selected Papers from the Third International Conference on Human-Computer Interaction (EWHCI '93).* Springer-Verlag, Berlin, Heidelberg, 210–218. http://dl.acm.org/citation.cfm?id=646181.682601

[25] Erik Løhre and Magne Jørgensen. 2016. Numerical Anchors and Their Strong Effects on Software Development Effort Estimates. *J. Syst. Softw.* 116, C (June 2016), 49–56. https://doi.org/10.1016/j.jss.2015.03.015

[26] Carolyn Mair and Martin Shepperd. 2011. Human Judgement and Software Metrics: Vision for the Future. In *Proceedings of the 2Nd International Workshop on Emerging Trends in Software Metrics (WETSoM '11).* ACM, New York, NY, USA, 81–84. https://doi.org/10.1145/1985374.1985393

[27] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2014. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories.* ACM, 192–201.

[28] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. 2016. An empirical study of the impact of modern code review practices on software quality. 21, 5 (2016), 2146–2189. https://doi.org/10.1007/s10664-015-9381-9

[29] Kannan Mohan and Radhika Jain. 2008. Using Traceability to Mitigate Cognitive Biases in Software Development. *Commun. ACM* 51, 9 (Sept. 2008), 110–114. https://doi.org/10.1145/1378727.1389970

[30] R. Mohanani, I. Salman, B. Turhan, P. Rodríguez, and P. Ralph. 2018. Cognitive Biases in Software Engineering: A Systematic Mapping Study. *IEEE Transactions on Software Engineering* (2018), 1–1. https://doi.org/10.1109/TSE.2018.2877759

[31] Don A. Moore and Paul J. Healy. 2008. The trouble with overconfidence. *Psychological Review* 115 (2008), 502–517.

[32] Eric Neumayer and Thomas Plümper. 2017. *Robustness tests for quantitative research.* Cambridge University Press.

[33] A. L. Nichols and J. K. Maner. 2008. The good subject effect: Investigating participant demand characteristics. *Journal of General Psychology* 135, 1 (2008), 151–165.

[34] Meike Olbrecht and Lutz Bornmann. 2010. Panel peer review of grant applications: what do we know from research in social psychology on judgment and decision-making in groups? *Research Evaluation* 19, 4 (10 2010), 293–304. https://doi.org/10.3152/095820210X12809191250762 arXiv:http://oup.prod.sis.lan/rev/article-pdf/19/4/293/4452351/19-4-293.pdf

[35] J. Parsons and C. Saunders. 2004. Cognitive heuristics in software engineering applying and extending anchoring and adjustment to artifact reuse. *IEEE Transactions on Software Engineering* 30, 12 (Dec 2004), 873–888. https://doi.org/10.1109/TSE.2004.94

[36] Jamie Peabody. [n.d.]. mergely. http://www.mergely.com/.

[37] Mitzi G Pitts and Glenn J Browne. 2007. Improving requirements elicitation: an empirical investigation of procedural prompts. *Information Systems Journal* 17, 1 (2007), 89–110. https://doi.org/10.1111/j.1365-2575.2006.00240.x

[38] Scott Plous. 1993. *The Psyhology of Judgement and Decision Making.* McGraw-Hill, Inc.

[39] Jason E Robbins and David F Redmiles. 1998. Software architecture critics in the Argo design environment. *Knowledge-Based Systems* 11, 1 (1998), 47 – 60. https://doi.org/10.1016/S0950-7051(98)00055-0

[40] Robert Brautigam. 2018. Why I Never Null-Check Parameters. https://dzone.com/articles/why-i-never-null-check-parameters.

[41] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern Code Review: A Case Study at Google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '18).* ACM, 181–190. https://doi.org/10.1145/3183519.3183525

[42] Scott Shipp. 2019. Better Null-Checking in Java. https://dev.to/scottshipp/better-null-checking-in-java-ngk.

[43] Davide Spadini. [n.d.]. CRExperiment. https://github.com/ishepard/CRExperiment.

[44] Davide Spadini, Maurício Aniche, Margaret-Anne Storey, Magiel Bruntink, and Alberto Bacchelli. 2018. When testing meets code review: Why and how developers review tests. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE).* IEEE, 677–687.

[45] Davide Spadini, Gül Çalikli, and Alberto Bacchelli. [n.d.]. Replication package for "Primers or Reminders? The Effects of Existing Review Comments on Code Review". https://doi.org/10.5281/zenodo.3653856.

[46] Davide Spadini, Fabio Palomba, Tobias Baum, Stefan Hanenberg, Magiel Bruntink, and Alberto Bacchelli. 2019. Test-Driven Code Review: An Empirical Study. In *Proceedings of the 41st International Conference on Software Engineering (ICSE2019).* Montreal, Canada, 1061–1072.

[47] Webb Stacy and Jean MacMillan. 1995. Cognitive Bias in Software Engineering. *Commun. ACM* 38, 6 (June 1995), 57–63. https://doi.org/10.1145/203241.203256

[48] KEITH E. STANOVICH. 2009. *What Intelligence Tests Miss: The Psychology of Rational Thought.* Yale University Press. http://www.jstor.org/stable/j.ctt1nq14j

[49] Patanamon Thongtanunam and Ahmed E Hassan. 2020. Review Dynamics and Their Impact on Software Quality. *IEEE Transactions on Software Engineering* (2020).

[50] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida. 2015. Investigating Code Review Practices in Defective Files: An Empirical Study of the Qt System. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories.* 168–179. https://doi.org/10.1109/MSR.2015.23

[51] Amos Tversky and Daniel Kahneman. 1973. Availability: A Heuristic for Judging Frequency and Probability. *Cognitive Psychology* 5 (1973), 207–232. https://doi.org/10.1109/TSE.2018.2877759

[52] Hans van Vliet and Antony Tang. 2016. Decision making in software architecture. *Journal of Systems and Software* 117 (2016), 638 – 644. https://doi.org/10.1016/j.jss.2016.01.017

[53] Alex Zhitnitsky. 2016. *The complete guide to Solving Java Application Errors in Production.* OverOps.