Test-Driven Code Review: An Empirical Study

Davide Spadini[1,2], Fabio Palomba[3], Tobias Baum[4]
Stefan Hanenberg[5], Magiel Bruntink[1] and Alberto Bacchelli[3]
[1]Software Improvement Group, The Netherlands - [2]Delft University of Technology, The Netherlands
[3]University of Zurich, Switzerlands - [4]Leibniz Universitat Hannover, Germany - [5]Universitat Duisburg-Essen, Germany

***Abstract*—Test-Driven Code Review (TDR) is a code review practice in which a reviewer inspects a patch by examining the changed test code *before* the changed production code. Although this practice has been mentioned positively by practitioners in informal literature and interviews, there is no systematic knowledge on its effects, prevalence, problems, and advantages.**

In this paper, we aim at empirically understanding whether this practice has an effect on code review effectiveness and how developers' perceive TDR. We conduct (i) a controlled experiment with 93 developers that perform more than 150 reviews, and (ii) 9 semi-structured interviews and a survey with 103 respondents to gather information on how TDR is perceived. Key results from the experiment show that developers adopting TDR find the same proportion of defects in production code, but more in test code, at the expenses of less maintainability issues in production code. Furthermore, we found that most developers prefer to review production code as they deem it more important and tests should follow from it. Moreover, widespread poor test code quality and no tool support hinder the adoption of TDR.

I. Introduction

Peer code review is a well-established and widely adopted practice aimed at maintaining and promoting software quality [3]. Contemporary code review, also known as *Change-based Code Review* [6] or *Modern Code Review* (MCR) [12], is a lightweight process that is (1) informal, (2) tool-based, (3) asynchronous, and (4) focused on inspecting new proposed code changes rather than the whole codebase [43]. Specifically, in a code review, developers other than the code change author manually inspect the patch to find as many issues as possible and provide feedbacks that need to be addressed before accepting the code in production [6].

The academic research community is conducting empirical studies to better understand the code review process [44], [43], [3], [26], [45], as well as to obtain empirical evidence on aspects and practices that are related to more efficient and effective reviews [51], [34].

A code review practice that has only been touched upon in academic literature [49], but has been described in gray literature almost ten years ago [58] is that of *test-driven code review* (TDR, henceforth). By following TDR, a reviewer inspects a patch by examining the changed test code *before* the changed production code.

To motivate TDR, P. Zembrod—Senior Software Engineer in Test at Google—explained in the Google Blog [58]: "When I look at new code or a code change, I ask: What is this about? What is it supposed to do? Questions that tests often have a good answer for. They expose interfaces and state use cases". Among the comments, also S. Freeman—one of the ideators of Mock objects [48] and TDD [9]—commented how he covered similar ground [21]. Recently, in a popular online forum for programmers, another article supported TDR (collecting more than 1,200 likes): "By looking at the requirements and checking them against the test cases, the developer can have a pretty good understanding of what the implementation should be like, what functionality it covers and if the developer omitted any use cases." Interviewed developers reported preferring to review test code first to better understanding the code change before looking for defects in production [49].

These above are compelling arguments in favor of TDR, yet we have no systematic knowledge on this practice: whether TDR is effective in finding defects during code review, how frequently it is used, and what are its potential problems and advantages beside review effectiveness. This knowledge can provide insights for both practitioners and researchers. Developers and project stakeholders can use empirical evidence about TDR effects, problems, and advantages to make informed decisions about when to adopt it. Researchers can focus their attention on the novel aspects of TDR and challenges reviewers face to inform future research.

In this paper, our goal is to obtain a deeper understanding of TDR. We do this by conducting an empirical study set up in two phases: An experiment, followed by an investigation of developers' practices and perceptions.

In the first phase, we study the effects of TDR in terms of the proportion of defects and maintainability issues found in a review. To this aim, we devise and analyze the results of an online experiment in which 92 developers (77 with at least two years of professional development experience) complete 154 reviews, using TDR or two alternative strategies (*i.e.*, production first or only production); also, two external developers rated the quality of their review comments. In the second phase, we investigate problems, advantages, and frequency of adoption of TDR – valuable aspects that could not be investigated in the experiment. To this aim, we conduct nine interviews with experiment participants and deploy an online survey with 103 respondents.

Key findings of our study include: with TDR, the proportion of functional defects (*bugs* henceforth) found in production code and maintainability issues (*issues* henceforth) found in test code does not change. However, TDR leads to the discovery of more bugs in test code, at the expenses of less issues found in production code. The external raters judged the quality of the review comments as comparable across all review strategies. Furthermore, most developers seem to be reluctant to devote much attention to tests, as they deem production code more important; moreover applying TDR is problematic, due to widespread poor test quality (reducing TDR's applicability) and no tool support (not easing TDR).

II. Related Work

To some extent, TDR can be considered as an evolution of classical reading techniques [4], as it shares the general idea to guide code inspectors with software artifacts (*i.e.*, test cases) and help them with the code review task.

Scenario-based inspections. Among reading techniques, Porter & Votta [39] defined the *scenario-based* approach, based on scenarios that provide inspectors with more specific instructions than a typical checklist and focus on a wider variety of defects. They discovered that such technique is significantly more useful for requirements inspectors. Later on, Porter *et al.* [40], [38] and Miller *et al.* [35] replicated the original study confirming the results. Other studies by Fusaro *et al.* [22] and Sandahl *et al.* [46] reported contradictory results, however without providing explanations on the circumstances leading scenario-based code inspection to fail. An important advance in this field was then provided by Basili *et al.* [5], who re-visited the original scenario-based as a technique that needs to be specialized for the specific issues to be analyzed. They also defined a new scenario-based technique called *perspective-based* reading: The basic idea is that different aspects of the source code should be inspected by inspectors having different skills [5]. All in all, the papers mentioned above provided evidence of the usefulness of reading techniques; their similarities with TDR, give an interesting rationale on why TDR could bring benefits.

Ordering of code changes. Research on the ordering of code changes is also related to TDR. In particular, Baum *et al.* argued that an optimal ordering of code changes would help reviewers by reducing the cognitive load and improving the alignment with their cognitive processes [8], even though they made no explicit reference to ordering tests. This may give theoretical value to the TDR practice. Code ordering and its relation to understanding, yet without explicit reference to tests or reviews, has also been the subject of studies [24], [10].

Reviewing test code. Many articles on classical inspection (*e.g.*, [31], [56]) underline the importance of reviewing tests; however, they do not leave any specific recommendation. The benefits of reviewing tests are also highlighted in two case studies [32], [37]. Already in Fagan's seminal paper [15], the inspection of tests is discussed, in this case noting fewer benefits compared to the inspection of production code. Winkler *et al.* [57] performed an experiment on writing tests during inspection and found neither large gains nor losses in efficiency and effectiveness. Elberzhager *et al.* [14], [13] proposed to use results from code reviews to focus testing efforts. To our knowledge, in academic literature TDR has been explicitly referred to only by Spadini *et al.* [49]. In a more general investigation on how test files are reviewed, the authors reported that some practitioners indeed prefer to review test code first as to get a better understanding of a code change before looking for defects in production code. Our work builds upon the research on reviewing test code, by investigating how reviewing test code can(not) be beneficial for the whole reviewing process.

III. Methodology

In this section we describe the research questions and the methodology we follow to conduct our study.

A. Research Questions

The overall goal of this paper is to obtain a deeper understanding of Test-Driven Code Review. This study has two parts that we structure in two research questions. In the first part, we start by designing and running an experiment to investigate the effects of TDR on code review effectiveness. We measure the effectiveness as the ability to find bugs and maintainability issues during a code review (*i.e.*, the main reported goal of code review [3]). This allows us to establish whether it is possible to empirically measure any significant difference in this aspect using TDR. Hence, our first research question:

> **RQ$_1$.** Does the order of presenting test code to the reviewer influence code review effectiveness?

In the second part of the study, we investigate the prominence of TDR and the developers' perception toward this practice, also focusing on problems and advantages that could be measured through the aforementioned experiment. Our aim is to obtain a more complete view on TDR. To do so, we turn to the developers, conducting semi-structured interviews and deploying an online survey. Hence, our second research question:

> **RQ$_2$.** How do developers perceive the practice of Test-Driven Code Review?

B. Method – *RQ$_1$*: Design Overview

Figure 1 depicts an overview of the overall flow of our experiment. We follow a partially counter-balanced repeated measures design [17], augmented with some additional phases.

1) We use a browser-based tool to conduct the experiment and answer **RQ$_1$**. The tool allows to (i) visualize and perform code reviews, and (ii) collect data from demographic-like questions and the interactions that participants have with the tool. The welcome page provides information on the experiment to perform and requires informed consent.

2) After the welcome page, an interface is shown to collect demographics as well as information about some confounding factors such as: (i) the main role of the participant in software development, (ii) Java programming experience, (iii) current practice in programming and reviewing, and (iv) the hours already worked in the day of the experiment to approximate the current mental freshness. These questions are asked with the aim of measuring real, relevant, and recent experience of participants, as recommended by previous work [16]. Once

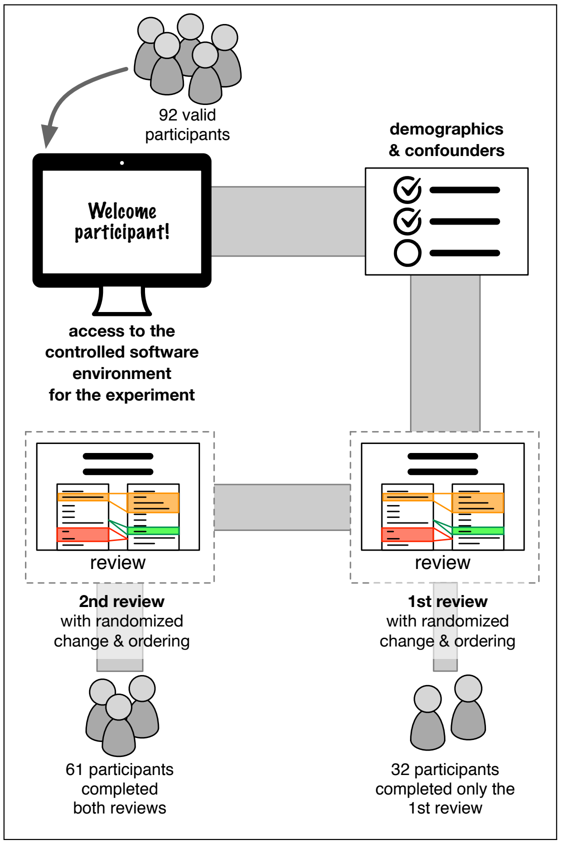

92 valid participants
Welcome participant!
access to the controlled software environment for the experiment
demographics & confounders
review
2nd review with randomized change & ordering
review
1st review with randomized change & ordering
61 participants completed both reviews
32 participants completed only the 1st review

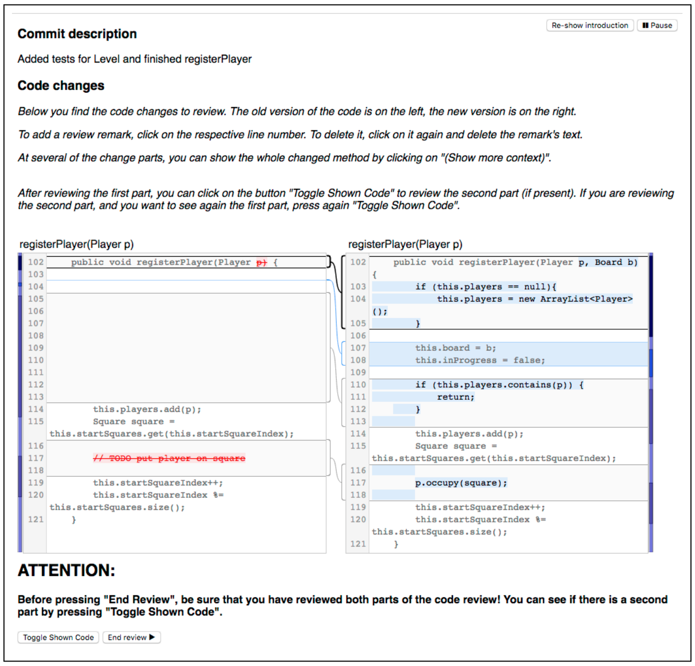
Figure 1. Experiment steps, flow, and participation

filled in this information, the participant receives more details on the reviews to be performed.

3) Each participant is then asked to perform two reviews (the first is mandatory, the second is optional), randomly selected from the following three treatments[1] that correspond to the TDR practice and its two opposite strategies:

- **TF** (*test-first*) – The participant must review the changes in both test code and production code, and is shown the changed test code first.
- **PF** (*production-first*) – The participant must review both production and test, and is shown the production code first.
- **OP** (*only-production*) – The participant is shown and must review only the changes to the production code.

For the treatments **TF** and **PF**, the tool shows a 'Toggle Shown Code' button that allows the participant to see and review the other part of the change (*e.g.*, the production code change if the treatment is **TF**). We do not limit the number of times the 'Toggle Shown Code' button can be clicked, thus allowing the participant to go back and forth from the test to the production code as desired. The participant can annotate review remarks directly from the GUI of our tool.



Commit description

Added tests for Level and finished registerPlayer

Code changes

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

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

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

After reviewing the first part, you can click on the button "Toggle Shown Code" to review the second part (if present). If you are reviewing the second part, and you want to see again the first part, press again "Toggle Shown Code".

registerPlayer(Player p)

```
    public void registerPlayer(Player ~~p)~~ {









        this.players.add(p);
        Square square =
this.startSquares.get(this.startSquareIndex);

        ~~// TODO put player on square~~

        this.startSquareIndex++;
        this.startSquareIndex %=
this.startSquares.size();
    }
```

registerPlayer(Player p)

```
    public void registerPlayer(Player p, Board b)
{
        if (this.players == null){
            this.players = new ArrayList<Player>
();
        }

        this.board = b;
        this.inProgress = false;

        if (this.players.contains(p)) {
            return;
        }

        this.players.add(p);
        Square square =
this.startSquares.get(this.startSquareIndex);

        p.occupy(square);

        this.startSquareIndex++;
        this.startSquareIndex %=
this.startSquares.size();
    }
```

ATTENTION:

Before pressing "End Review", be sure that you have reviewed both parts of the code review! You can see if there is a second part by pressing "Toggle Shown Code".

Figure 2. Example of the review view in the browser-based experiment UI, showing the code change. In this case, the treatment is **PF**, thus to see the test code, the participant must click on the 'Toggle Shown Code' button.

4) Before submitting the experiment, we ask to the participants if they would like to be further contacted for a semi-structured interview; if so, they fill in a text field with their email address. Further comments/impressions on the study can be reported using a text block.

*C. Method – **RQ**$_1$: Browser-Based Experiment Platform*

As previously mentioned, we adapt and use a browser-based experiment platform to run the experiment. This has two main advantages: On the one hand, participants can conveniently perform code reviews; on the other hand, the tool assists us when gathering data from the demographic questions, conducting the different treatments, and collecting information for the analysis of co-factors. To reduce the risk of data loss and corruption, almost no data processing is done on the server: instead, participants' data is recorded as log records and analyzed offline.

The tool implements a GUI similar to other browser-based code review tools (*e.g.*, GITHUB pull requests): It presents a code change in form of two-pane diffs. An example of the GUI implemented is reported in Figure 2: Review remarks can be added and changed by clicking on the margin beneath the code. The tool logs a number of user interactions, such as mouse clicks and pressed keys, which we use to ensure that the participants are actively performing the tasks.

*D. Method – **RQ**$_1$: Objects*

The objects of the study are represented by the code changes (or *patch*, for brevity) to review, which need to be properly

[1] Our choice of proposing two of the three treatments is driven by our willingness of keeping the experiment as short as possible to stimulate a high response rate, as also recommended by Flanigan *et al.* [18]; nevertheless, this does not influence our observations as the random selection guarantees a balance in the answers among all treatments (see Table IV), thus allowing us to effectively study their effects on code review performance and quality.

selected and eventually modified to have a sufficient number of defects to be discovered by participants.

Patches. To avoid giving some developers an advantage, we use a code base that is not known to all the participants of the experiment. This increases the difficulty of performing the reviews. To keep the task manageable, we ensure that (i) the functional domain and requirements are well-known to the participants and (ii) there is little reliance on special technologies or libraries. To satisfy these goals, we select an open-source project, **NAME BLINDED**:[2] The project consists of 42 production classes (≈2k LOC) as well as 13 test classes (≈600 LOC), it received 103 pull requests in his history, and has a total of 17 contributors.

To select suitable patches, we screen the commits of the project and manually select changes that (1) are self-contained, (2) involve both test and production code, (3) are neither too complicated nor too trivial, and (4) have a minimum quality, *e.g.*, not containing dubious changes.

The selected patches are those of commits **BLINDED** and **BLINDED**. In the first one, a new feature is added along with the tests that cover it. In the second one, a refactoring in the production code is applied and a new test is added.

Seeding of bugs and maintainability issues. Code review is employed by many software development teams to reach different goals, but mainly (1) detecting bugs (functional defects) and (2) improving code quality (*e.g.*, finding maintainability issues), (3) spreading knowledge [7], [33], [51], [3].

Since the online experiment is done by single developers, we measure code review effectiveness by considering only the first two points, detecting bugs (functional defects) and maintainability issues. We seed more bugs and maintainability issues. Examples of injected bugs are a wrong copy-paste and wrong boundary checking. For maintainability issues, we mainly mean *"smells that do not fail the tests"* [19], *e.g.*, a function that does more than what it was supposed to do, wrong documentation or variable naming. At the end, the two patches contain a total of 4 bugs and 2 issues (Patch 1) and 5 bugs and 3 issues (Patch 2).

E. Method – ***RQ***$_1$*: Variables and analysis*

We investigate whether the proportion of defects found in a review is influenced by the review being done under a **TF**, **PF**, or **OP** treatment, controlling for other characteristics.

The first author of this paper manually analyzed all the remarks added by the participants. Our tool explicitly asked and continuously highlighted to the participants that the main goal is to find both bugs and maintanability issues; therefore, each participant's remark is classified as identifying either a bug or an issue, or as being outside of the study's scope. A remark is counted only if in the right position and correctly pinpointing the problem.[3]

Table I
VARIABLES USED IN THE STATISTICAL MODEL

Metric	Description
Dependent Variables	
ProdBugsProp	Proportion of functional defects found in the *production* code
ProdMaintIssuesProp	Proportion of maintainability issues found in the *production* code
TestBugsProp	Proportion of functional defects found in the *test* code
TestMaintIssuesProp	Proportion of maintainability issues found in the *test* code
Independent Variable	
Treatment	Type of the treatment (TF, PF, or OP)
Control Variables	
Review Details	
TotalDuration	Time spent in reviewing the code
IsFirstReview	Boolean representing whether the review is the first or the second
Patch	Patch 1 or 2
Profile	
Role	Role(†) of the participant
ReviewPractice	How often(†) they perform code review
ProgramPractice	How often(†) they program
ProfDevExp	Years of experience(†) as professional developer
JavaExp	Years of experience(†) in Java
WorkedHours	Hours the participant worked before performing the experiment

(†) see Table III for the scale

By employing values at defect level, we could compute the dependent variables at review level, namely proportions given by the ratio between the number of defects found and the total number of defects in the code (dependent vars in Table I). The dependent variables are then given by the average of n_j binary variables y_i, assuming a value 1 if the defect is found and 0 if not, where n_j is the total number of defects present in the change j, so that the proportion π_j results from n_j independent events of defect finding and y_j are binary variables that can be modeled through a logistic regression.

$$\pi_j = \sum_{1}^{n_j} \frac{y_j}{n_j} \tag{1}$$

The main independent variable of our experiment is the review strategy (or *treatment*). We consider the other variables as control variables, which include the time spent on the review, the review and programming practice, the participant's role, the reviewed patch (*i.e.*, P1 or P2), and whether the review is the first or the second being performed. In fact, previous research suggest the presence of a trade-off between speed and quality in code reviews [25]; following this line, we

[2] Due to double blind restrictions, we cannot reveal the project's name. However, to support the review of our paper, we include an anonymized snapshot of the system in our replication package, also including the patches selected for the experiment and the seeded defects [1].

[3] To validate this coding process, a second author independently re-coded the remarks and compared his classification with the original one. In case of disagreements, the two authors opened a discussion and reached a consensus. We compute the Cohen's kappa coefficient [11] to measure the inter-rater agreement between the two authors before discussion: we find it to reach 0.9, considerably higher than the recommended threshold of 0.7 [11]. The detailed coding of all review remarks can be found in our online appendix [1].

expect longer reviews to find more defects; to check this, we do not fix any time for review, allowing participants to perform the task as long as needed. Moreover, it is reasonable to assume that participants who perform reviews more frequently to also find a higher share of defects.

We run logistic regressions of proportions, where $Logit(\pi_j)$ represents the explained proportion of found defects in review j, β_0 represents the log odds of being a defect found for a review adopting **PF** (or **OP**) and of mean TotalDuration, IsFirstReview, *etc.*, while parameters $\beta_1 \cdot Treatment_j$, $\beta_2 \cdot TotalDuration_j$, $\beta_3 \cdot IsFirstReview_j$, $\beta_4 \cdot Patch_j$, *etc.* represent the differentials in the log odds of being a defect found for a change reviewed with **TF**, for a review with characteristics $TotalDuration_{j-mean}$, $IsFirstReview_{j-mean}$, $Patch_{j-mean}$, *etc.*.

$$\begin{aligned} Logit(\pi_j) = \beta_0 + \beta_1 \cdot Treatment_j + \beta_2 \cdot TotalDuration_j + \\ + \beta_3 \cdot IsFirstReview_j + \beta_4 \cdot Patch_j + \\ + ...(other\ vars\ and\ \beta\ omitted) \end{aligned} \tag{2}$$

*F. Method – **RQ**$_2$: Data Collection And Analysis*

While through the experiment we are able to collect data on the effectiveness of TDR, the perception of the developers on the prevalence of TDR as well as the motivations for applying it or not cannot be collected. Hence, to answer **RQ**$_2$, we proceed with two parallel analyses: We (i) perform semi-structured interviews with participants of the experiment who are available to further discuss on TDR and (ii) run an online survey with the aim of receiving opinions from the broader audience of developers external to the experiment.

Semi-structured interviews. We design an interview whose goal is to collect developers' points of view on TDR. They are conducted by the first author of this paper and are semi-structured, a form of interview often used in exploratory investigations to understand phenomena and seek new insights on the problem of interest [55].

Each interview starts with general questions about code reviews, with the aim of understanding why the interviewee performs code reviews, whether they consider it an important practice, and how they perform them. Then, we ask participants what are the main steps they take when reviewing, starting from reading the commit message to the final decision of merging/rejecting a patch, focusing especially on the order of reviewing files. Up to this point, the interviewees are not aware of the main goal of the experiment they participated in and our study: We do not reveal them to mitigate biases in their responses. After these general questions, we reveal the goal of the experiment and we ask their personal opinions regarding TDR. The interview protocol is available [1].

During each interview, the researcher summarizes the answers and, before finalizing the meeting, these summaries are presented to the interviewee to validate our interpretation of their opinions. We conduct all interviews via SKYPE. With the participants' consent, the interviews are recorded and transcribed for analysis. We analyze the interviews by initially assigning codes [28] to all relevant pieces of information, then group these codes into higher-level themes, which we discuss in our results (Section V).

Table II
INTERVIEWEES' EXPERIENCE (IN YEARS) AND WORKING CONTEXT

ID	Developer	Reviewer	Working context	Applying TDR
P1	8	8	OSS	Almost never
P2	3	3	Company A	Almost Always
P3	15	15	Company A	Almost Always
P4	10	10	Company B	Almost Never
P5	10	5	Company C	Always
P6	3	2	Company D	Sometimes
P7	16	16	Company E	Always
P8	4	4	Company F / OSS	Sometimes
P9	3	3	Company G / OSS	Never

Overall, we conduct nine interviews (each one 20/30 minutes long); Table II summarizes the demographical information about the participants.

Online survey. We create an anonymous online survey (requiring approximately 4 minutes to be filled out), organized into two sections. In the first one, we ask demographic information of the participants, including gender, programming/reviewing experience, policies regarding code reviews in their team (*e.g.*, if all changes are subject of review or just a part of them), and whether they actually review test files (the respondents who answer "no" are disqualified). In the second section, we ask (i) how often they start reviewing from tests and how often from production files and (ii) request them to fill out a text box explaining the reasons why they start from test/production files. The questionnaire is created using a professional tool named SURVEYGIZMO[4] and is spread out through practitioners blogs (*e.g.*, REDDIT) and through direct contacts in the professional network of the study authors, as well as the authors' social media accounts on Twitter and Facebook. As a result, we collect 103 valid answers, which are used to complement the semi-structured interviews. Among the respondents, 5% have one year or less of development experience, 44% have between 2 to 5 years, 28% between 6 and 10 years, and 23% more than 10 years.

IV. RESULTS – **RQ**$_1$: ON THE EFFECTS OF TDR

We present the results of the experiment we run to measure the effects of TDR on finding defects during a code review.

A. Collected data, filtering, and considered cases

A total of 232 people accessed our experiment enviroment following the provided link. From their reviews (if any), 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 5 minutes spent on the review. We also remove an outlier review that lasted more than 4 standard deviations from the mean review time, without entering any comments. After applying the exclusion criteria, a total of 92 participants remained for the subsequent analyses. Table III presents what the participants reported in terms of role, experience, and practice.

[4] https://www.surveygizmo.com

Table III
PARTICIPANTS' CHARACTERISTICS – DESCRIPTIVE STATISTICS - N. 92

	Dev	**Student**	**Researcher**	**Architect**	**Analyst**	**Other**
Current role	61% (56)	16% (15)	12% (11)	5% (5)	3% (3)	2% (2)
Experience (years) with	**None**	**<= 1**	**2**	**3-5**	**6-10**	**>10**
- Java prog.	13% (12)	5% (5)	7% (6)	21% (19)	34% (31)	21% (19)
- Profess. dev.	5% (5)	11% (10)	13% (12)	18% (17)	28% (26)	24% (22)
Current frequency of	**Never**	**Yearly**	**Monthly**	**Weekly**	**Daily**	
- Programming	0% (0)	0% (0)	3% (3)	17% (16)	79% (73)	
- Reviewing	15% (14)	7% (6)	16% (15)	22% (20)	40% (37)	

Table IV
DISTRIBUTION OF PARTICIPANTS' REVIEWS ACROSS TREATMENTS

	TF	**PF**	**OP**	*total*
Patch1	31	32	29	*92*
Patch2	28	29	34	*91*
total	*59*	*60*	*63*	

Table V
AVERAGE PROPORTION OF BUGS AND ISSUES FOUND, BY TREATMENT, AND REVIEW TIME. COLORED COLUMNS INDICATE A STATISTICALLY SIGNIFICANT DIFFERENCE BETWEEN THE TREATMENTS ($p < 0.01$), WITH THE COLOR INTENSITY INDICATING THE DIRECTION.

	Proportion of found				**Time**
	production		**test**		
	bugs	**maintIssues**	**bugs**	**maintIssues**	
TF	0.28	**0.08**	**0.40**	0.18	7m11s
PF	0.33	**0.21**	**0.17**	0.13	6m27s
OP	0.28	**0.18**			5m29s

Table VI
REGRESSIONS FOR 'PRODMAINTISSUESPROP' AND 'TESTBUGSPROP'

	TestBugsProp			**ProdMaintIssuesProp**		
	Estimate	S.E.	Sig.	Estimate	S.E.	Sig.
Intercept	-0.7314	1.9242		-0.0471	1.2191	
TotalDuration	0.1664	0.0618	**	0.0462	0.0259	.
IsFirstReview 'TRUE'	-0.9213	0.5344	.	-0.1554	0.2241	
Patch 'P2'	1.9296	0.5688	***	-2.8474	0.4579	***
Treatment 'PF'				0.0975	0.2386	
Treatment 'TF'	1.1792	0.4639	**	-1.2468	0.3908	**
ReviewPractice	0.0675	0.2082		0.2598	0.1389	.
ProgramPractice	-0.6608	0.4685		-0.2180	0.2938	
ProfDevExp	-0.0982	0.1951		-0.2953	0.1211	*
JavaExp	-0.0182	0.1512		0.0366	0.0631	
WorkedHours	-0.0225	0.0817		0.0403	0.0405	
... (†)						

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

Only 5 of the participants reported to have no experience in professional software development; most program daily (79%) and review code at least weekly (62%). Table IV shows how the participants' reviews are distributed across the considered treatments and by reviewed patch. Despite some participants completed only one review and the aforementioned exclusions, the automated assignment algorithm allowed us to obtain a rather balanced number of reviews per treatment and by patch.

B. Experiment results

Table V shows the average values achieved by the reviews for the dependent variables (*e.g.*, 'ProdBugsProp') and the average review time, by treatment. The most evident differences between the treatments are in: (d1) the proportion of maintainability issues found in production code (**PF** and **OP** have an average of .21 and .18, respectively, while **TF** of 0.08), and (d2) the proportion of bugs found in test code (**TF** has an average of 0.40, while **PF** of 0.17). Indeed, running the Wilcoxon Signed Rank Test [29] (which is a non-parametric test that makes no assumptions on the underlying data distribution) we find that these two differences are statistically significant ($p < 0.01$) and the effect size (Cliff's delta) is medium (d1) and small (d2). On the contrary, all the other differences are not statistically significant (the minimum p is higher than 0.38). In other words, we find initial evidence that **PF** and **OP** may provide a significant gain in detecting maintainability issues within the production code (*i.e.*, the first code that is shown), when compared to **TF**. At the same time, the average proportion of defects found in test code is significantly higher for the reviews done with **TF**, when compared to **PF** (the **OP** treatment cannot be compared because it did involve any test code). We do not find any statistically significant difference in the other outcomes (*i.e.*, proportion of defects in production and proportion of maintainability issues in test). Also, we do not find any significant difference in the time taken for reviews with **TF** vs. those with **PF**: Reviewers consume nearly the same amount of time in both the cases.[5] Overall, the comparison of the averages highlights that, within the same time, developers who started reviewing from tests (**TF**) spot a similar number of bugs in production, while discovering more defects in the tests but fewer maintainability issues in the production code. Thus, there seems to be a compromise between reviewing test or production code first when considering defects and maintainability issues.

With regression modeling, we investigate whether these differences are confirmed when taking into account the characteristics of participants and reviews (variables in Table I). In Section VI we describe the steps we take to verify that the selected regression model is appropriate for the available data.

We build the four models corresponding to the four dependent variables, independently. Confirming the results shown in Table V, the treatment is statistically significant exclusively for the models with 'ProdMaintIssuesProp' and 'TestBugsProp' as dependent variables; Table VI reports the results.[6] We observe that—also considering the other factors—**TF** is confirmed a statistical significant variable in both 'ProdMaintIssuesProp' and 'TestBugsProp', with negative and positive directions, respectively. To calculate the odds of being a maintainability issue found in a review with **TF** compared to the baseline **OP**, we exponentiate the differential logit, thus: $exp(-1.25) = 0.29$, which means 71% less chances to find the issue in case of **TF** than **OP** (or **PF**). Instead, the odds of being a test bug found in a review with **TF** is 3.49, thus almost 250% more chances to find the test bug than with **PF**. Also, we see that

[5] **OP**'s time is only reported for description and is not comparable to the other treatments as only production code is reviewed in this treatment.

[6] For space reasons, we omit the other two models, in which no variable is significant, but these models are available in our online appendix [1].

the specific patch under review has a very strong significance for both models, thus confirming that differences in the code are an essential factor in the final outcome. The review time plays a significant role for 'TestBugsProp' and (to a lesser degree) for 'ProdMaintIssuesProp', in the expected direction. Unexpectedly, the professional experience plays a negative role for 'ProdMaintIssuesProp'; we hypothesize that professionals focus more on functional defects than maintainability issues.

> ***Finding 1****. In reviews in which tests are presented first, participants found significantly more bugs in test code, yet significantly fewer maintainability issues in production code. The production bugs and test maintainability issues found is stable across the treatments.*

C. Assessing code review quality

After having found differences in bugs/issues found with **TF**, we check whether the different treatments (**TF**, **PF**, **OP**) have an influence on the quality of the review: to this aim two external validators manually classify each review, rating each comment. These validators have more than 5 years of industrial experience in code review and have collaborated in many open-source projects. We request them to go through each of the code reviews done by the developers involved in the experiment and rate each comment aimed at fixing a defect or maintainability issue in the production or test code using a Likert scale ranging between '1' (very poor comment) and '5' (very useful comment) [7]. They also give a score to each comment that is outside the scope (*i.e.*, a comment that does not fix any of our manually injected defects), plus a final overall score for the review. Each validator performs the task independently and then their assessments is sent back to the authors of this paper. The validators are unaware of which treatment was used in each review. A set of 33 reviews is classified by both authors, so that we can measure their inter-rate agreement using Cohen's kappa [11]. To mitigate the personal variability of the raters, we cluster their ratings into three categories: 'below average review comment', 'average review comment' and 'good review comment'. Then, we check the raters' agreement. The result shows that in all ratings there is at least a substantial agreement between the validators: production issues ($\kappa = .69$), test issues ($\kappa = .78$), production bugs ($\kappa = 1$), test bugs ($\kappa = .77$).

To measure the influence of a treatment on the review quality we apply the ANOVA test on the independent variables score (*e.g.*, productivity issues) and the dependent variable `treatment`. As a result, we find that for no score the independent variable is a significant factor (productivity issues p=.62; test issues p=.30; productivity bugs p=.25; test bugs p=.37); thus, we conclude that raters do not see a difference in the quality of the reviews for different treatments.

> ***Finding 2****. There is no statistically significant difference between the quality of the reviews made under the three considered treatments, according to two external raters blinded to the underlying treatments.*

V. Results – $\mathbf{RQ}_2$: On The Perception Of TDR

In $\mathbf{RQ}_1$ we presented an investigation on the effectiveness of TDR. However, the developers' perception on the practice, the motivations to apply it, and the frequency of adoption cannot be investigated by means of an online experiment. For this reason, we discuss with some of the participants of the experiment what they perceive as advantages and disadvantages of TDR. We also survey 103 developers to enrich the data collection with people that do not participate in the experiment and can provide a complementary view on TDR. In this section, we report the answers obtained during our interviews and surveys. We summarize them in topics, covering both advantages and disadvantages of this practice. We refer to the interviewees by their ID shown in Table II.

A. Adoption of TDR *in practice*

When analyzing the survey data, 5% (5) of the respondents indicated that they always start from test code, 13% (13) almost always, 42% (43) occasionally/sometimes, 27% (28) almost never, and 13% (13) never. Thus, the majority of the respondents indicate that they occasionally apply TDR. This gives even more importance to this investigation, since there is not empirical knowledge on why developers do it, and what are the main advantages/disadvantages of this practice.

B. Perceived problems with TDR

When analyzing data coming from surveys and interviews, we discover a set of blocking points for the adoption of TDR. They can be grouped around four main themes, *i.e.*, perceived importance of tests, knowledge gained by starting reviewing from tests, test code quality, and code review tool ordering, that we further discuss in the following. Themes are discussed based on the frequency of appearance in the survey.

Tests are perceived as less important. From the comments left by the participants of the survey, it seems clear that test code is considered much less important than production code and that, as stated by one participant, they *"want to see the real change first"*. This strong opinion is confirmed by other 15 participants;[8] for example, a participant explains that s/he starts *"looking at production files in order to understand what is being changed about the behavior of the system. [S/he] views tests as checks that the system is behaving correctly so it does not seem possible to evaluate the quality of the tests until I have a clear understanding of the behavior"*. While the semi-structured interviews confirmed this general perception around tests, they also add a more practical point to the discussion:

[7] We chose a 5-Points Likert scale because it is rather not probable that raters can agree on a more fine-grained scale

[8] It should be noted that in the survey we gave the possibility to leave open comments; having 15 or more participants agreeing on exactly the same theme indicates an important trend.

$P_{1,2-5,7}$ state that they need to prioritize tasks because of time, and often higher priority is given to the production code.

A closely related factor contributing to this aspect is the tiredness associated with reviewing code for a long time. As reported by one of the survey participants, *"the longer you are reviewing the more sloppy you get [...]. I would rather have a carefully reviewed production file with sloppy test than vice versa"*. In other words, when performing multiple code review at once, developers often prefer to pay more attention to the production code than test files.

13 participants also report that is the production code to *drive* tests rather than the opposite. A clear example of this concept is enclosed in the following participant's quote: *"To me, tests are about checking that a piece of software is behaving correctly, so it doesn't make sense to me to try to understand if the tests are testing the right conditions if I do not understand what the code is supposed to do first."* Finally, another aspect influencing the perception that developers have of tests is lack of testing experience. One of the participants affirms that *"not everyone in my team has lots of experience with testing, so usually just looking at the production code will tell me if there will be problems with the tests."* Thus, having poor experience in testing practice might bias the perception of the advantages given by tests in the context of code review.

Tests give less knowledge on the production behavior. Both interviewees and survey respondents report that the main advantage of starting with production code is that they can immediately see the feature and how it is changed, and only later they will check if it is properly tested. For example, a survey respondent says: *"I want to understand what the production code does, form my own opinion of what should be tested, and then look at the tests afterward."* From the interviews, P_9 also adds that it is hard to capture all the possible behaviors with tests, while looking to production code first helps him/her figuring out the failure modes before seeing what the tests the developer proposed are.

Nevertheless, an interesting trend emerges from our results. Despite most developers claim that tests cannot give enough knowledge to review a change, six of them declare that the decision of start reviewing from test code basically depends on the *degree* of knowledge they have of the production code under test. As explained by a survey participant, *"If I am familiar with the topic and the code, I will start with the production files. Otherwise I will choose the test files"*. In other words, tests only seem to be useful in time of need, *i.e.*, when a developer does not have any other instrument to figure out the context of the proposed code change.

Tests have low code quality. 4 participants mention poor test code quality as a reason to not apply TDR in practice. The use of tests in a code review clearly has the prerequisite that such tests can properly exercise the behavior of the production code. One of the participants, when explaining why s/he prefers starting from production code, reports that *"sometimes the tests are bad and it is easier to understand how the code behave by looking at the feature code"*. This is also confirmed by the semi-structured interviews: the main obstacle when reviewing tests first is the assumption that the test code is well written [$P_{1-3,5,6,9}$]. Both P_1 and P_9 said that most of the times they prefer to start from production because they assume developers do not write good tests. P_1 says *"Usually, I start from production and maybe the reason is that many of the projects where I worked do not have that many tests"*. Even if the tests are present, sometimes reviewers find them difficult to understand: According to P_6, *"... the test needs to be written in a clear way to actually understand what's going on."* The solution to this problem—according to our interviewees—is to impose test rules, *e.g.*, all the changes to the production code should be accompanied by tests [$P_{2,3,5}$]. P_5 says that *"if [tests] are not good, I will ask to modify them. If they are not even present, I will ask to add them and only after I will review the patch"*. change in their IDE and run the tests before start reviewing.

Code review tool ordering. The final disadvantage is related to a practical problem: current code review tools present code changes following an alphabetic order, meaning that most of the times developers review following such order. This is highlighted by 15 survey participants and confirmed by the interviewees. For example, P_3 says: *"If there is a front-end, we do not have integration tests for all the features, so sometimes I do manual test. In this case I would stick with the order of GitLab. I think GitLab present tests later than production, so I generally start from that"*. Interestingly, this point came up also from 35% of the survey respondents who do not apply TDR, that indicated they start from production code because they simply follow the order given by the code review tool. According to interviewees P_3 and P_4, they follow the order of GitHub because in this way they are sure to have reviewed all the files in the patch, while going back and forth from file to file may result in skipping some files.

> ***Finding 3**. Perceived problems with* TDR*: Developers report to (1) consider tests as less important than production, (2) not being able to extract enough knowledge from tests, (3) not being able to start a review from tests of poor quality, and (4) being comfortably used to read the patch as presented by their code review tool.*

C. Perceived advantages of TDR

We identify two main themes representing the major perceived advantages of adopting TDR, *i.e.*, the coincise, high-level overview tests give on the functionalities of production code and the ability of naturally improving test code quality. We discuss those aspects based on the frequency in the survey.

TDR provides a black-box view of production code. 18 of the respondents explain that the main advantage they envision from the application of TDR is the ability of tests to provide a coincise, high-level overview of the functionalities implemented in the production code. In particular, one participant reports: *"If I read the test first without looking* *at the production implementation, I can be sure that the test describes the interface clearly enough that it can serve as documentation."* Moreover, developers appreciate that few lines of test code allows them to contextualize the change. Most interviewees agreed that starting reviewing tests allows them to understand better the code change context [P_{2-8}]. P_3 says: *"I think starting from tests helps you in understanding the context first, the design, the "what are we building here?" before actually looking at "how they implemented it"*. The common feeling between the interviewees is that tests better explain what the code is supposed to do, while the production code shows how the developer implemented it [$P_{1-4,6,7}$]. P_1 says: *"When you are reviewing complex algorithms it is nice to immediately see what type of outputs it produces in response to specific types of input."* P_2 adds that it is easier to understand what the developer implemented, because through the tests a reviewer can easily see what the code is supposed to return given a specific input, and this is not always clear looking at the production code.

TDR improves test code quality. Three of the survey participants explicitly report that tests must be of good quality and a practice like TDR would enable a continuous test code quality improvement; as one put it: *"Tests are often the best documentation for how the production code is expected to function. Getting tests right first also contributes to good TDD practices and the architectural values that come with that"*. In other words, the developers report that, in situations where reviewers inspect tests first, the improvement of test code quality have to necessarily happen, otherwise reviewers could not properly use tests as documentation to spot problems in production code. As a natural consequence, TDR would also enforce tests to be updated, thus producing overall benefits to the system reliability. Furthermore, one participant reports TDR to be efficient *"because it captures and should capture all the bugs"*: even if we obtain this kind of feedback by a small number of developers, it is still interesting to remark how some of them perceive the potential benefits of TDR, which we empirically found (**RQ**$_1$), as being more effective in terms of test bugs discovered—while spotting the same proportion of production bugs. The semi-structured interviews confirm all the aspects discussed so far. Most of the interviewees agreed that TDR somehow helps reviewers to be more focused on testing. According to P_9: *"I think it would encourage the development of good tests, and I think better tests mean more bugs captured. So yes, at the end you may capture more bugs"*. P_6 and P_8 also say that when reviewing the production code the reviewer already knows what the code is tested for, so it could be *easier* to catch not covered paths. For example, P_3 refers to happy *vs.* bad paths: *"starting from the tests you think a little bit on the cases that apply, so for example if they only test the happy path and not the bad path"*.

***Finding 4**. Perceived problems with TDR: Developers report that* TDR *(1) allows them to have a concise, high-level overview of the code under test and (2) helps them in being more testing-oriented, hence improving the overall test code quality.*

VI. Threats to validity

In this section we describe the threats to validity and limitations to the results of our work, as posed by the research methodology that we applied.

Construct validity Threats to construct validity concern our research instruments. Many of the constructs we use are defined in previous publications and we reuse existing instruments as much as possible: The tool employed for the online experiment is based on a similar tool used in previous work [1].

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; only afterward we started the release phase.

One of the central measures in our study is the number of defects and maintainability issues found. The defects were seeded by the first author, and later checked by the other authors. Nevertheless, we cannot rule out implicit bias in seeding the defects as well as in selecting the code changes.

We asked the participants to review test and production code separately, using a "Toggle shown code" button to switch between them. However, we cannot ensure that all the participants used this button correctly (or used it at all). To mitigate this, we analyzed the results mining only the shown code (*e.g.*, the second part of the review would not exist), obtaining very similar results: Hence we can conclude that this threat is not affecting the final results.

A major threat is that the artificial experiment created by us could differ from a real world scenario. We mitigated this issue in multiple ways: (1) we used real changes, (2) we reminded the participant to reviewing the files as they would normally do in their daily life, and (3) we used an interface very similar to the common Code Review Tools GUIs.

Furthermore, to validate the reviews done by the participants, we involved two external validators. The only information they had at their disposal when rating the reviews was the patch and the comments of the participants, *i.e.*, they did not know what treatment it was, the duration of the review, and all the other information we collected. Thus, the rate given by the validators was based on their personal judge and past experience in code review. To mitigate this issue, we involved two validators that have strong experience in Java and in MCR: one had worked for many years in a large Russian-based SE company, and the other validator holds a PhD in SE and have worked in many OSS.

Internal validity - Credibility Threats to internal validity concern factors we did not consider that could affect the variables and the relations being investigated. In an online setting, a possible threat 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. Furthermore, to exclude participants that did not take seriously the experiment, we excluded experiments without any comments in the review, and we did a manual classification of the comments to delete the inappropriate ones.

We do not know the nature of the population that did our experiment, hence it might suffer from a self-selection bias. Indeed, it could be possible that the sample contains better and more motivated reviewers than the population of all software developers. However, we do not believe this poses a major risk to the validity of our main findings since we would expect stronger effects with a more representative sample. Furthermore, as depicted in Table IV, the participants experience is quite various, with a 30% lower then 2 years and 50% with more than 6.

External validity - Transferability Threats to external validity concern the generalization of results. A sample of 93 professional software developers is quite large in comparison to many experiments in software engineering [47]. However, it is still small compared to the overall population of software developers that employ MCR. We reduce this issue by interviewing and collecting the opinions of other developers that did not participate in the experiment and surveying 103 developers.

Statistical conclusion validity 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. Even though we reached a quite large sample of participants, our sample is not large enough to detect small effects for RQ_1, hence we might not be able to reliable calculate some statistics.

A major threat to our RQ_1 results is to employ the wrong statistical model. 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 1.5 (values should be below 10), thus indicating little or no multicollinearity among the independent variables, (2) run a multilevel regression model [42] 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, (3) ascertain the linearity (assumed by logistic regression) of our independent continuous variable (the review time) and log odds using the Box-Tidwell test [23], and (4) build the models by adding the independent variables step-by-step and found that the coefficients remained stable, thus further indicating little to no interference among the variables.

Finally, in our statistical model we control for the type of the patch, namely Patch 1 or 2. However, we do not control for Product or Process metrics of the code (*i.e.*, size, complexity, churn, etc.): we control only for the patch as it encloses all the characteristics that previous literature already demonstrated as related to review effectiveness [33], [3].

VII. Discussion and Implications

Our findings provide two key observations to be further discussed and that lead to implications for practitioners, educators, tool vendors, and research community.

Ordering of files within the code review. Interviewees and survey respondents indicated that they often review the files as presented by their own code review tool. While this process has the advantage that at the end a reviewer is sure to have reviewed all files, this order is not always the correct one: for example, our experiment (**$\mathbf{RQ}_1$**) showed that looking first at tests allows a reviewer to capture more test code bugs, which have been shown to be extremely harmful for the overall reliability of software systems [36], [50], [54]. At the same time, TDR still allows to catch the same amount of bugs in production code, thus being nearly equivalent to the case of reviewing production files first. However, the drawback consists of finding less issues in production.

We also found that developers decide on whether to start reviewing from test or production based on different factors such as familiarity with the code or type of modification applied. This suggests that, to improve productivity and code review performance, tool vendors might enable the option to let developers decide on the code review ordering. At the same time, the research community is called to the definition of novel techniques that can exploit a set of metrics (*e.g.*, change type or past modifications of the developer on the code under review) to automatically recommend the order that would allow the reviewer to be more effective: this would lead to the definition of new adaptive mechanisms that take into account developer-related factors to improve the reviewability of source code [41].

Test code quality. On the other hand, the main enemy of TDR seems to be the poor quality of test code [2], [53]. Many interviewees and survey respondents indicated this as the main reasons to *not* apply TDR. If tests are badly written or incomplete it becomes almost impossible (or even dangerous) to start reviewing from test code, as it is harder to spot errors in production code. If a possible solution consists of enforcing the introduction of code of conducts that explicitly indicate rules on how to review tests [52], our findings motivate and encourage the steadily growing research area around test code quality management and improvement [20], [27], [30].

At the same time, the development of a good team culture in which test code is considered as important as production code should be a must for educators. Indeed, as previous work already pointed out [3], [33], [49], good reviewing effectiveness is found mostly within teams that value the time spent on code review; hence, practitioners should set aside sufficient time for reviewing all the files present in the patch, including test code.

VIII. Conclusion

We assessed (i) whether the order of presenting test or production code to reviewers influences their code review performance and (ii) the developers' perception concerning TDR, by performing an experiment with 92 developers, 9 semi-structured interviews and a survey study with 103 participants. We found that the proportion of production bugs and test issues does not change, however TDR leads to the discovery of more bugs in test code at the expenses of less issues found in production code. Then, we discover that the application of TDR can be problematic because of poor test quality, time pressure, and no tool support. These findings represent the main inputs for our future research agenda, which is focused on defining novel techniques and tools to ease TDR in practice.

References

[1] Test-Driven Code Review: An Empirical Study - Online Appendix. http://www.mediafire.com/folder/3b5ey849y9flx/Test-Driven_Code_Review_-_Online_Appendix, 2018.

[2] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman. Test code quality and its relation to issue handling performance. *IEEE Transactions on Software Engineering*, 40(11):1100–1125, 2014.

[3] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *35th International Conference on Software Engineering*, ICSE 2013a, pages 710–719, 2013.

[4] V. Basili, G. Caldiera, F. Lanubile, and F. Shull. Studies on reading techniques. In *Proc. of the Twenty-First Annual Software Engineering Workshop*, volume 96, page 002, 1996.

[5] V. R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sørumgård, and M. V. Zelkowitz. The empirical investigation of perspective-based reading. *Empirical Software Engineering*, 1(2):133–164, 1996.

[6] T. Baum, O. Liskin, K. Niklas, and K. Schneider. A faceted classification scheme for change-based industrial code review processes. In *Software Quality, Reliability and Security (QRS), 2016 IEEE International Conference on*, Vienna, Austria, 2016. IEEE.

[7] T. Baum, O. Liskin, K. Niklas, and K. Schneider. Factors influencing code review processes in industry. In *Proceedings of the ACM SIGSOFT 24th International Symposium on the Foundations of Software Engineering*, Seattle, WA, USA, 2016. ACM.

[8] T. Baum, K. Schneider, and A. Bacchelli. On the optimal order of reading source code changes for review. In *33rd IEEE International Conference on Software Maintenance and Evolution (ICSME), Proceedings*, 2017.

[9] K. Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[10] B. Biegel, F. Beck, W. Hornig, and S. Diehl. The order of things: How developers sort fields and methods. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 88–97. IEEE, 2012.

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

[12] J. Cohen. Modern code review. In A. Oram and G. Wilson, editors, *Making Software*, chapter 18, pages 329–338. O'Reilly, 2010.

[13] F. Elberzhager, J. Münch, and D. Assmann. Analyzing the relationships between inspections and testing to provide a software testing focus. *Information and Software Technology*, 56(7):793–806, 2014.

[14] F. Elberzhager, A. Rosbach, J. Münch, and R. Eschbach. Inspection and test process integration based on explicit test prioritization strategies. In *Software Quality. Process Automation in Software Development*, pages 181–192. Springer, 2012.

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

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

[17] A. Field and G. Hole. *How to design and report experiments*. Sage, 2002.

[18] T. S. Flanigan, E. McFarlane, and S. Cook. Conducting survey research among physicians and other medical professionals: a review of current literature. In *Proceedings of the Survey Research Methods Section, American Statistical Association*, volume 1, pages 4136–47, 2008.

[19] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[20] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013.

[21] S. Freeman. Sustainable Test-Driven Development. https://www.infoq.com/presentations/Sustainable-Test-Driven-Development, May 2010.

[22] P. Fusaro, F. Lanubile, and G. Visaggio. A replicated experiment to assess requirements inspection techniques. *Empirical Software Engineering*, 2(1):39–57, 1997.

[23] G. Garson. Logistic regression: binary & multinomial: 2016 edition (statistical associates "blue book" series). *Asheboro, NC: Statistical Associates Publishers*, 2016.

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

[25] T. Gilb and D. Graham. *Software Inspection*. Addison-Wesley, 1993.

[26] G. Gousios, M. Pinzger, and A. v. Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, pages 345–355, Hyderabad, India, 2014. ACM.

[27] M. Greiler, A. Van Deursen, and M.-A. Storey. Automated detection of test fixture strategies and smells. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 322–331. IEEE, 2013.

[28] B. Hanington and B. Martin. *Universal methods of design: 100 ways to research complex problems, develop innovative ideas, and design effective solutions*. Rockport Publishers, 2012.

[29] M. R. Hess and J. D. Kromrey. Robust Confidence Intervals for Effect Sizes: A Comparative Study of Cohen's and Cliff's Delta Unter Nonnormality and Heterogeneous Variances. *PhD Proposal*, 1:1–30, 2015.

[30] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2011.

[31] O. Laitenberger and J.-M. DeBaud. An encompassing life cycle centric survey of software inspection. *Journal of Systems and Software*, 50(1):5–31, 2000.

[32] F. Lanubile and T. Mallardo. Inspecting automated test code: a preliminary study. In *Agile Processes in Software Engineering and Extreme Programming*, pages 115–122. Springer, 2007.

[33] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. 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*, pages 192–201. ACM, 2014.

[34] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. An empirical study of the impact of modern code review practices on software quality. 21(5):2146–2189, 2016.

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

[36] F. Palomba and A. Zaidman. Does refactoring of test smells induce fixing flaky tests? In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 1–12. IEEE, 2017.

[37] O. Petunova and S. Bērziša. Test case review processes in software testing. *Information Technology and Management Science*, 20(1):48–53, 2017.

[38] A. Porter and L. Votta. Comparing detection methods for software requirements inspections: A replication using professional subjects. *Empirical software engineering*, 3(4):355–379, 1998.

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

[40] A. A. Porter, L. G. Votta, and V. R. Basili. Comparing detection methods for software requirements inspections: A replicated experiment. *IEEE Transactions on software Engineering*, 21(6):563–575, 1995.

[41] A. Ram, A. A. Sawant, M. Castelluccio, and P. . i. T. . W. Y. . . Bacchelli, Alberto. Booktitle = 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, FSE 2018, Lake Buena Vista, Florida, November 4-9, 2018.

[42] S. W. Raudenbush and A. S. Bryk. *Hierarchical linear models: Applications and data analysis methods*, volume 1. Sage, 2002.

[43] P. C. Rigby and C. Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 202–212, Saint Petersburg, Russia, 2013. ACM.

[44] P. C. Rigby, B. Cleary, F. Painchaud, M. Storey, and D. M. German. Contemporary peer review in action: Lessons from open source development. *Software, IEEE*, 29(6):56–61, 2012.

[45] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli. Modern code review: a case study at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pages 181–190. ACM, 2018.

[46] K. Sandahl, O. Blomkvist, J. Karlsson, C. Krysander, M. Lindvall, and N. Ohlsson. An extended replication of an experiment for assessing methods for software requirements inspections. *Empirical Software Engineering*, 3(4):327–354, 1998.

[47] D. I. Sjøberg, J. E. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanovic, N.-K. Liborg, and A. C. Rekdal. A survey of controlled experiments in software engineering. *Software Engineering, IEEE Transactions on*, 31(9):733–753, 2005.

[48] D. Spadini, M. Aniche, M. Bruntink, and A. Bacchelli. To mock or not to mock? an empirical study on mocking practices. In *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*, pages 402–412. IEEE, 2017.

[49] D. Spadini, M. Aniche, M.-A. Storey, M. Bruntink, and A. Bacchelli. When testing meets code review: Why and how developers review tests. In *Software Engineering (ICSE), 2018 IEEE/ACM 40th International Conference on*, page to appear, 2018.

[50] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli. On the relation of test smells to software code quality. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME). IEEE*, 2018.

[51] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida. Investigating code review practices in defective files: An empirical study of the qt system. In *MSR '15 Proceedings of the 12th Working Conference on Mining Software Repositories*, 2015.

[52] P. Tourani, B. Adams, and A. Serebrenik. Code of conduct in open source projects. In *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*, pages 24–33. IEEE, 2017.

[53] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. An empirical investigation into the nature of test smells. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, pages 4–15. IEEE, 2016.

[54] A. Vahabzadeh, A. M. Fard, and A. Mesbah. An empirical study of bugs in test code. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 101–110. IEEE, 2015.

[55] R. S. Weiss. *Learning from Strangers: The Art and Method of Qualitative Interview Studies*. 1995.

[56] E. F. Weller. Lessons from three years of inspection data (software development). *Software, IEEE*, 10(5):38–45, 1993.

[57] D. Winkler, S. Biffl, and K. Faderl. Investigating the temporal behavior of defect detection in software inspection and inspection-based testing. In *International Conference on Product Focused Software Process Improvement*, pages 17–31. Springer, 2010.

[58] P. Zembrod. Test-Driven Code Review. https://testing.googleblog.com/2010/08/test-driven-code-review.html, 2010.