Wojciech Frącz
Jacek Dajda

SOURCE CODE REVIEWS ON MOBILE DEVICES

Abstract   *This paper presents the results of an experiment-driven investigation on the efficiency of source code review practice performed on mobile devices. In particular, the conducted investigation tries to verify whether or not the small screens of mobile devices influence the speed and quality of the review process. Besides presenting the experiment itself and discussing the obtained results, this paper also describes the dedicated Android application for mobile code reviews that was implemented for research purposes.*

1. Introduction

The practice of source code reviewing is one of the techniques utilized by modern software development teams in order to maintain code quality. While some tools are available (e.g., Checkstyle for Java or JSHint for Javascript programming languages), regular code reviews (in its various forms) still seem unbeatable in the context of code readability and clarity [1].

As with every practice, this technique has its flaws. Probably the greatest is the lack of proper process support and management encouragement. Most of the software methods developed in the past 20 years (including the popular Agile Methods) do not include source code reviews as a core practice. One of the few exceptions is eXtreme Programming that introduces a controversial practice of Pair Programming, which is meant to sustain a process of continuous code review. Fortunately, current trends in the software industry seem favorable for code review by providing tools such as Gerrit [8] or technique of *Pull Requests*, both of which not only technically facilitate the review but also impose it into the development process.

This paper contributes to these trends with the idea of mobile code reviews performed on mobile devices such as a smartphone, or better, a tablet. Doing a review while sitting on a comfortable sofa in a company social space seem definitely more attractive to the average software developer than a typical review performed at his desk. Another important benefit is the higher communication comfort. It is much easier to discuss specific code fragments with nearby developers just by passing them the device instead of inviting them to the reviewer's desk. In addition, mobile code review can make the practice more available during business trips, especially when the offline mode is available. These aspects seem vital in encouraging developers to bring more focus to code quality and frequent reviews.

On the other hand, several questions arise regarding whether or not the physical limitations of mobile devices (in particular, small screens and uncomfortable virtual keyboards) can affect the quality and efficiency of such reviews, not to mention other aspects (such as developer fatigue and general attitude). This paper provides preliminary answers to these questions by presenting early findings on mobile reviews based on the experimental comparison of mobile and classic desktop-based reviews.

The paper is organized as follows: the next section briefly introduces code reviews and gives a summary of the related work and current software support. Section 4 describes the applied research method, including the presentation of the developed tool and organized evaluation experiments. Section 5 provides the experimental results with corresponding charts (which are commented upon in the next section). The final section summarizes the conducted research and introduces plans for future work.

2. The practice of source code review

The origins of code review practice can be found in the formal inspections proposed by M. Fagan [2]. Fagan defined inspection as a stage-based process that can be applied

not only to source code but also by other artifacts of every software project, including requirement specification, system design, or even documentation. The process is driven by meetings (repeated endlessly until the artifacts are considered complete), which are attended by team members assigned to specific roles (including authors, reviewers, and moderators). According to Fagan, code inspections require us to spare 54 hours for each 1000 lines of code. These claims are confirmed by other, much more contemporary studies as well [9, 1]. Code inspections are also praised [10] for their learning and knowledge sharing effects, which greatly speed up development and reduce project risk (in case core developers leave the team).

The benefits of the inspections are valuable, but they do not come without a cost. Such a process as most formalized approaches must be expensive. [9, 4] indicate costs as one of the main reasons for which inspections are not widely used in software development. To cut costs, developers experiment with more lightweight variants of Fagan's inspections. This results in different types of inspections leading to new terminology [1], in which *inspections* are replaced by *code review* or *peer review*.

The more lightweight and nowadays-common peer reviews do not assume any phases of development. They are instead performed in a *code-assess-respond* style. Each piece of work that is done must be reviewed by a peer – strictly speaking, another developer on a team. This approach is more flexible and can be adjusted to internal team processes and preferences. Even though peer reviews are less expensive and more flexible than formal Fagan inspections, they are not widely used either. [9] suggests that the fundamental reason for this may be the fact that they are not the most enjoyable engineering task when compared to design and coding.

It is also worth mentioning that maintaining code quality is not the only use of code reviews. This technique can also be used to estimate existing code nature. Such estimations are invaluable when it comes to legacy code.

Defects injection is one of the code quality estimating techniques [3]. The method starts by selecting a legacy code system fragment. One developer analyzes it to understand it completely. Then, he introduces (injects) a few defects into the code. They must not be obvious, as their aim is to compare injected and existing source code problems. The next step is to give the "enhanced" version of the code to another developer. After reviewing it, he found some defects that were artificially created and some that were in the code before the injection of defects. Having these numbers, we can calculate the approximate number of defects in the reviewed source code as

$$f = \frac{E \cdot D}{I} \tag{1}$$

f – estimated number of defects in a fragment
E – the number of existing (not injected) defects found in review
D – the number of defects injected by the first developer
I – the number of injected defects found in review

For example, if $D = 10$ new defects are injected into the source code fragment and the second developer finds $I = 8$ of them and $E = 3$ others, then analyzed code

is expected to have one more existing flaw.

$$f = \frac{3 \cdot 10}{8} = 3.75 \approx 4 \qquad (2)$$

Having estimated one fragment of a legacy system source code, the rest of it might be evaluated further. Comparing the size of the chosen fragment to the size of the whole system, the expected total number of bugs can be estimated as follows:

$$F = \frac{f}{l} \cdot L \qquad (3)$$

f – estimated number of defects in a fragment
l – length of a fragment (lines of code)
L – length of the whole system

If the fragment from the example above had 800 lines and was chosen from a 200k-line system, the expected number of bugs in all of the software would be

$$F = \frac{3.75}{800} \cdot 200000 = 937.5 \qquad (4)$$

This technique is strongly based on the assumption that a representative fragment of a system source code contains approximately the same amount of defects as any other fragment with the same length. Therefore, special care must be taken when choosing the software source code part for the estimation process. Of course, it is difficult to evaluate the accuracy of this technique (no research on its evaluation is reported in literature), however it gives some insights on how the quality can be measured.

3. Code review tools

Over recent years, the software industry has developed a more positive approach to code reviews. To make the developer's life easier, several dedicated tools are widely used, such as Gerrit [8], Attlasian Stash, CodeRemarks[1], and others. They facilitate the process by displaying code with an ability to review it by publishing comments. Added remarks are sent to the code's author, who can respond to them and fix the mistakes.

Both Gerrit and Attlasian Stash ease the code review process by allowing the creation of many versions of one logical change to the source code. The author changes the code to complete the issue on which he is currently working. Modifications are published to the selected reviewer in order to have them assessed. When the reviewer flags some of them, he writes his remarks as text comments. They are then returned back to the author. This creates the possibility of increasing code quality until the peer review is successful.

---

[1]`http://www.coderemarks.com`

Another major benefit of using such tools is that they ensure that no unchecked code sneaks into the production version of a piece of software. Source code without a review is blocked until a designated reviewer approves or rejects it. Obviously, this has the disadvantage of slowing work down, but it comes with a major gain of being certain about system quality.

Specialized code review applications are not the only choice these days. Patches or pull requests (PR) from a version control system can be subject of reviews too. Version control systems like Git [5] or SVN allow us to create excerpts with a list of changes to be made to the code. Such a list can be reviewed before it is merged into the code, incorporating all code review benefits.

This technique is often used in open source software. The vast majority of freeware source code libraries have their repositories publicly available. Whenever a developer need an improvement to such a library, he can suggest the author introducing it. Whether or not the change will be made is completely up to the library developer. When time matters, there is often no possibility to wait for his decision. It is a place for creating a PR. An outside developer makes the modifications he desires and sends it to the library owner. Actually, it changes the developer situation, as he is not asking somebody to do the work for him. He shows his effort by having the work already finished and asks the author only to introduce his changes in the next library release.

PRs can and should be reviewed before they reach the open source project repository. It can be done either by the source code owner or a community. There are many platforms that support creating and reviewing PRs, including Github[2] and Bitbucket[3]. The major flaw when using this method is the process of improving PRs when they do not pass the review. In such a situation, the author of the PR is supposed to create a new PR with all of the detected defects fixed. This process results in creating series of subsequent PRs, which are difficult to compare with each other and analyze.

Another trend in code review tools is a community peer review. It can be widely used even by single-user development teams or freelancers. The author publishes source code to review on the Internet. It can be either a publicly available forum or any other web application that allows the publication of source code (for example, CodeRemarks – as mentioned before). The only requirement for successful community review is to have volunteers who are willing to read through the published code and assess it. This is not as difficult as it seems to be. Developers may expect a decent code review in less that one hour when posting the code on one of the most popular community review designed platforms – Code Review Stack Exchange[4].

It seems that the proper tooling support improves the *enjoyment factor* of the reviews, and developers are more keen to perform it. Thus, this paper proposes that we go one step further: to support code reviews on mobile devices that are common these

---

[2]http://github.com
[3]http://bitbucket.org
[4]http://codereview.stackexchange.com

days. It is observed that all existing solutions are designed for the typical developer environment, which is desktop or laptop. On the other hand, code review is more about reading than writing and, therefore, can have more in common with studying a book than coding. Taking the developer out of his usual workplace to a social room or comfortable sofa should have a positive impact on his/her attitude to this practice. However, a question arises whether or not mobile code review will be as efficient as if performed on a desktop. This paper aims at providing preliminary experimental findings in this matter. No other similar or even related research is reported in the literature so far.

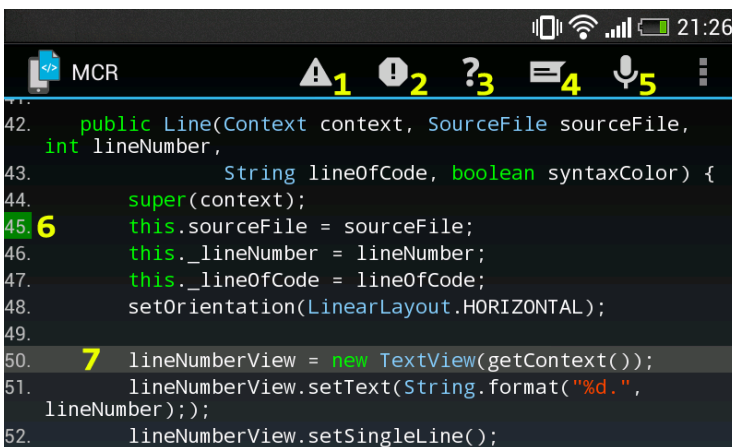## 4. Experimental comparison of mobile and desktop reviews

To examine the effectiveness of mobile reviews, a comparison of mobile and desktop reviews needs to be performed. That is why the applied research method consists of the two following elements:

1. dedicated review tool for mobile device created for the purpose of the research,
2. experimental comparison of the results obtained from mobile reviews and desktop reviews.

   Both elements are described in detail in the consecutive subsections.

4.1. Tool for mobile code reviews

The developed tool is targeted towards the Android platform (the most popular platform in Poland at the moment). This popularity was crucial for experimental purposes to make sure we will got a good balance between mobile and desktop reviews. A screenshot of the working application is presented in Figure 1.

Figure 1. Source code displayed in mobile application.

The most important functions can be summarized as follows:

- displaying source code with syntax coloring and line numbering,
- selecting lines (see item #7 in Figure 1) and attaching comment to it (by using buttons from #1 to #5 in Figure 1),
- marking lines that have comments (green background of line number – see item #6 in Figure 1),
- sharing comments with others.

To overcome the lack of a physical keyboard and improve the comfort of a reviewer's work, some innovations have been introduced to the presented solution. The most important one is predefined comments. This feature enables the reviewer to add comments with one-finger touch on top of the application menu (buttons #1 to #3 in Figure 1). A list of the predefined comments that were available during the experiment is presented in Table 1. In addition, it is possible to record voice comments and hook them to specific lines of code (button #5 in Figure 1). These features are created in order to benefit as much as possible from the capabilities of modern mobile devices. They also minimize the inconvenience of typing in the comments when using a virtual on-screen keyboard (button #4 in Figure 1).

Table 1

List of predefined comments.

Button #1	Button #2	Button #3
Magic number	Syntax error	Typo
Find a better name	Code duplication	Too complicated
Extract constant	Unhandled exception	Use existing library
Extract method	Unused argument / variable	Useless comment
Introduce explaining variable	Unused code	Invalid format

As for sharing comments, they are distributed in a ZIP archive that contains added remarks in JSON files. They can be sent with any Android service that allows file sharing – including e-mail, Bluetooth, or Wi-Fi. When such a file is being opened with the created tool, comments are automatically extracted and displayed in the source code.

4.2. Experimental comparison

The prepared experiment was designed specifically for comparison between classic (desktop) and mobile reviews.

4.2.1. Participants

The participants were Computer Science students from their third and fourth years of a BSc Studies degree from the University of Science and Technology. The study involved 55 programmers, 23 of whom performed code review using their own mobile devices (mainly smartphones). This allowed us to obtain some diversity in screen resolution and inspect its influence on work comfort.

All participants were familiarized with code smells detection and review process. They were also fluent with the Java language, which is the main language taught in the Computer Science Department at the University of Science and Technology. The choice of Java was therefore deliberate, as this allowed us to minimize the risk of situations in which participants fail the review task due to a lack of syntax or semantic knowledge of a language. As for the utilized tools, both of them (desktop and mobile) were a novelty to all participants. Also, a short introduction and demo of tool handling were performed before the experiment.

To conclude, we put an emphasis on obtaining comparable conditions for both desktop and mobile reviews: participants at the same level of education and random distribution among the two tested tools and environments.

4.2.2. Compared tools

Each participant performed a code review either on a mobile device or a PC computer using a CodeRemarks online tool for code review. The decision whether a particular student should use a mobile device or PC was left to him/herself so as not to impose a specific environment in which he/she could feel uncomfortable. Both mobile phones and tablets were allowed.

CodeRemarks was chosen for desktop reviews because it does not need any installation or configuration on a reviewer's device. It offers features comparable to the mobile application (display one source file, add text comments). Each reviewer was given a unique URL address where he/she could review the code. It made both distributing the task and collecting the results simple.

Obviously, a developed Android application was prepared for the experiment needs. The prepared version could only display source code that was the subject of study. After the timeout had been reached, the application automatically sent comments to the author, including the screen size and orientation of the device when the code review was performed.

4.2.3. Timing

Organizing volunteers for an experiment is always a problem for researches. Therefore, to obtain a reasonable number of participants, we decided to carry out the review task during student class time. This enforced a strict time limit. Based on a few initial (testing) reviews without a timeout, it was decided that seven minutes should suffice for the needs of the preliminary evaluation. It is far less time that should be dedicated for a review of such code but should be enough for an initial comparison of the PC and mobile code review efficiency.

4.2.4. Gathered data

The following data was gathered for each participant:
- list of review comments including line number, type, and content of the comment,
- timestamp of comment creation,

- screen resolution (for Android reviews only),
- screen orientation (for Android reviews only),
- review duration.

4.2.5. Task

Each participant obtained the same task, which was a prepared Java class to be reviewed. The source code of the Java class that was the subject of the study is presented in Appendix 1. The code is a fragment of a real program and was deliberately "enhanced" to contain more defects (code smells). As a result, 120 lines of code contain 30 code smells, 26 of which were known before the experiment (the remaining 4 were identified by the participants). The injected code smells were based on a list published in *Clean Code* [6] as well as on the authors' experience.

5. Results

The experiment reviews were collected and analyzed after the experiment had been finished. Each of the reviewer's comments was labelled as *valuable* (when it pointed to a recognized code smell) or *worthless* (when the comment could not be understood or did not point to any code smell). Several different aspects were inspected during experiment analysis. The most interesting ones are presented here.

5.1. The number of comments

The first aspect is the average total number of comments added in one review session. The average number of comments per review for Android tool is 8, while for desktop, it is below 6. However, when only valuable comments are taken into consideration, the difference is much smaller and fluctuates around 4.5 comments per review. Quantities are visualized in Figure 2.
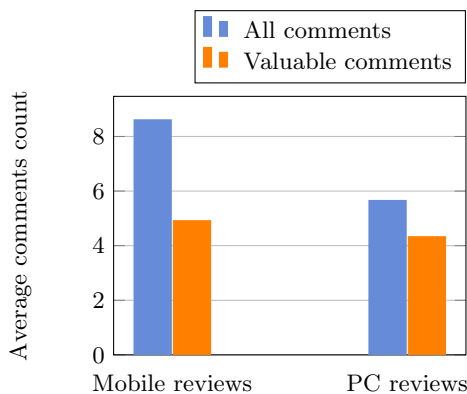

Figure 2. Average number of comments.

While the number of detected code smells in one review is comparable for both types of review, it cannot be argued that mobile devices introduce unnecessary clutter by many worthless remarks. Only 57% of mobile review comments are considered valuable.

It might be expected that the usability of mobile applications for code reviews is relatively small when compared to a large PC screen. In order to verify this statement, we inspected the number of comments that were misplaced (i.e., assigned to a wrong line number, but the comment's content would still allow the programmer to understand it).

The results do not differ much in either type of reviews. There are 9.3% and 9.9% misplaced remarks in mobile and PC reviews, respectively.

5.2. Mobile application usage

The majority of examined Android reviewers performed reviews in vertical screen orientation (75.8%). Devices that were held vertically are tablets with high screen resolution. They still allowed the display of whole lines of code without breaking them.

We tried to investigate the relationship between screen resolution and comment quantity or value, but the results have not revealed any pattern.

As for the type of comments that were attached to the source code, their frequency is visualized in Figure 3. The vast majority of students used the predefined comments feature. There were a few participants that added voice comment.

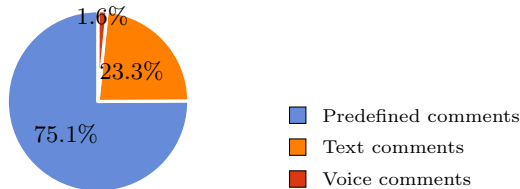**Figure 3.** Type of comments added when using mobile application.

5.3. Detected code smells

Four of the code smells that had been injected into the reviewed source code were not found. These smells are:

- long list of imports not shortened with wildcard import (smell #2 from Appendix 2),
- overridden safety by unnecessary definition of `serialVersionUID` (#4)
- flag argument (#10),
- hidden temporal coupling (#14).

However, the reviewers managed to find four new flaws that were unknown at the time of preparing the task:

- *"reinventing the wheel"* (smell #21 from from Appendix 2),
- inconsistent code formatting (#25 and #27),
- no defensive copy (#29).

The most conspicuous flaws were found the most frequently. Three of the most recognizable code smells are shown in Table 2.

Table 2

Code defects with the highest detection rate.

Smell no.	Defect	Detection (mobile)	Detection (PC)
#6	unnecessary field name encoding	61%	47%
#24	commented out code	52%	31%
#18	ambiguous method name	43%	66%

Smells #13, #21 and #29 (see Appendix 2) were found only in PC reviews. However, PC reviewers were not able to find #3, #23 and #25 (which mobile reviewers managed to do).

In order to find further differences between PC and mobile reviews, we also analyzed the order in which reviewers found defects. This might reveal a pattern in which reviews are performed with each tool. We aimed at distinguishing elements that were paid attention to more in mobile reviews than in PC reviews. Figure 4 shows how many times a particular smell was found as the first one.
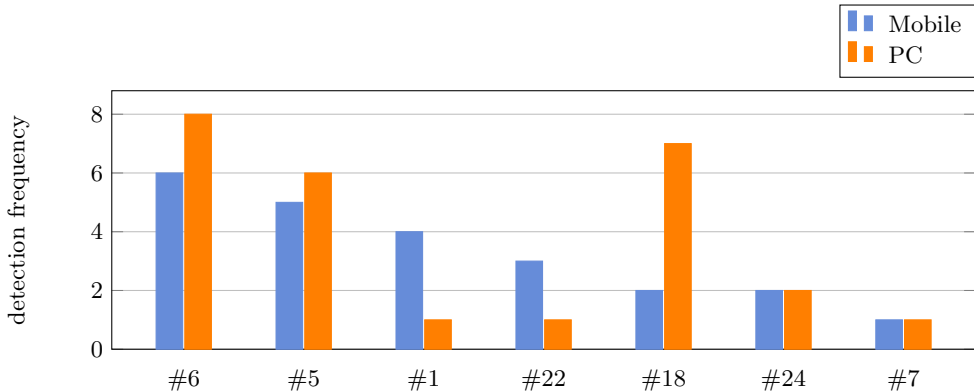

Figure 4. The first detected smells in reviews.

Surprisingly, there are only 7 *first-smells* on mobile code reviews, whereas on PCs, there are 12. *First-smells* that were detected only in PC reviews are #11 (3 times) and #12, #17, #20, #30 (once).

6. Discussion

The summarized results presented in Section 5 allow us to make several interesting observations.

6.1. Code reviews are effective regardless of the method of their performing

When we ignore worthless comments, it occurs that the quantity of found code smells in one session is similar for both PC and mobile reviews. This means that mobile reviews are not worse than PC reviews.

As for the efficiency of code reviews, the average number of code smells found in both types of reviews is equal to 4.68. Comparing it to the total number of known defects of the reviewed code, we get the amount of 15.6% code smells found in one review.

[7] suggests that code reviewers are expected to find more than 50% of defects in the code during one session. Our results are much worse; however, when a time limit is taken into consideration, it is clear that the reviewers were not able to find the expected number of code smells. The most efficient reviews need to last 88 minutes per 188 lines of code [7]. Therefore, a reviewer would need 56 minutes for 120 lines of code. The experiment allowed for only 7 minutes, so we can calculate the expected number of defects found in the conducted study as:

$$x = \frac{7}{56} \approx 13\% \tag{5}$$

The value is much more appropriate now and proves both code reviews' efficiency and the described ideal speed of performing them.

6.2. Mobile code reviews are not uncomfortable

This statement is formed mainly based on almost the same frequency of misplaced comments in PC and mobile reviews. The main reason for gathering such information about the remarks added to the code was the belief that results of mobile reviews would contain many comments that were added to a wrong line. However, the performed analysis did not prove this expectation. It turns out that it is equally easy to tap the wrong line on a touch display as to click it on a PC screen. With the predefined and voice comments available, mobile application can be an even more comfortable review tool to use.

However, it needs mentioning that the method of displaying the source code to the reviewer is crucial. The vast majority of respondents used horizontal screen orientation during their reviews. Only high-resolution mobile devices allowed them to perform these vertically (tablets). Undoubtedly, it is harder to read source code when its lines are wrapped. Therefore, such results had been expected. Still, they led to the conclusion that allowing more customizations of code appearance would increase tool usability.

It cannot be overlooked that mobile reviewers added more than 50% more comments to the code than the PC reviewers. This may be the result of trying out the application. There were comments like "test" that introduced nothing into the review process. Also, predefined comments may encourage a reviewer to add meaningless remarks, as it is so easy to add them. Therefore, they must be selected very carefully.

6.3. Predefined comments are promising

Predefined comments were most frequently added when using a mobile application. It is surprising that such a feature is not common in existing tools supporting code reviews. Participants were excited about the possibility of adding a comment to the code by a single tap. In this way, they can make the review much quicker and in a more comfortable manner.

There were cases when a predefined comment added to the code did not address a code smell exactly. As an example, predefined comment *"Find a better name"* added to the line with the useless `TWO = 2` constant does not clearly express the intention of the reviewer on how to improve the code. However, there is a high probability that the author of the code reading such a remark would think about this *poor naming* comment, which should lead to its removal (or renaming). Therefore, such comments in the study were marked as valuable.

During the experiment, the application had the predefined comments hardcoded. Many participants suggested allowing the ability to add their own predefined comments on the basis of each project. Such enhancement would allow us to adjust vocabulary and common types of defects to the reviewer, which would make them even more useful.

6.4. Voice comments introduce communication problems

There were only three voice comments added by three participants in the experiment. They were nothing more than simple "fix it" phrases. This undoubtedly indicates some problems with this technique.

Firstly, the way of performing the study should be taken into account. Most of the reviews were done simultaneously in a group of students. Recording a voice comment in public can be perceived as weird behavior when everybody hears what a reviewer thinks about a particular fragment of a code. Moreover, if each participant would start to record a voice comment, they would be hard to understand.

It seems that this can also affect real developers who work in co-located teams. However, for freelance, open-source, or distributed developers, this way of performing reviews can provide some practical value.

6.5. Small screen enhances details

Careful analysis of smells that were found in mobile reviews only lead to the conclusion that displaying source code on small screens forces reviewers to pay more attention to details of the code. Tiny code defects like #23 (typographic error – `recodedFile`

instead of `recordedFile`) or #25 (lack of space before curly brace) were found only in mobile reviews. When seeing only a few lines of code at a time, reviewers can focus more on such trivialities (which is harder to do on a PC).

On the other hand, the small screen prevents reviewers from seeing the whole picture of a code being reviewed. Semantic error in code (#13) was found only during PC reviews. It required a detailed analysis of the entire class, so it was hard to detect on mobile devices.

6.6. Small screen encourages reviewers to analyze code line by line

The first code smells that were detected reveal a pattern that was used to perform code reviews on both devices. Defects placed in the beginning of the file (lines from 1 to 48) were found as first-smells in over 70% of mobile reviews and in almost 50% of PC reviews. This indicates that reviewing with a mobile application encourages the analysis of source code line by line.

6.7. *Defects injection* method is working

The method for code quality estimation described in the beginning of the article is effective. The source code used in the experiment had been very carefully analyzed by authors; yet, four new defects have been found by the respondents. This indicates that adding artificial bugs to the code and letting two or more people check it is valuable, indeed.

The conducted research results in 26 smells injected into the code, 21 of which were found during the reviews. 4 smells are new. Using Formula 1, the expected quantity of fragment defects can be obtained.

$$f = \frac{4 \cdot 26}{21} \approx 5 \tag{6}$$

This result suggests that there is one more flaw in the presented fragment of code that has yet to be found. It also allows us to calculate the estimated number of defects in the application from which the source code was taken (Formula 3).

$$F = \frac{5}{120} \cdot 1000 \approx 42 \text{ / 1000 lines of code} \tag{7}$$

7. Conclusions and future work

The aim of this research was to inspect the aspect of code reviews in the context of mobile devices. There are two important conclusions that need to be drawn from this research.

First of all, mobile reviews have been proven to be as efficient as the classic (desktop) code reviews we do these days. The developed prototype Android tool for code reviews as well as thr organized experiment prove that, in some aspects, mobile reviews can be even more convenient than desktop reviews. Having agreed that the

comfort and *enjoyment factor* are important for developers [9], we believe that mobile reviews will find their practical application in the software industry in the near future.

Secondly, the conducted study confirmed that code reviews are really an effective way of ensuring the quality of a source code. Almost all of the code smells that were prepared for the experiment were found, and the collected comments were assessed as correct and helpful in fixing the errors and making the code cleaner and more readable.

Further research should include the organization of la onger experiment with more participants involved in order to confirm the preliminary results and observations. An interesting aspect would also be to compare both types of reviews in the context of a complex code structure. In this way, the observation that the small screen of a mobile device limits the reviewer in obtaining the bigger picture of the source code could be verified.

What is more, it is planned to deploy the prototype Android tool in a real development environment. A good candidate is a project developed at University of Science and Technology for Government Protection Bureau (funded by the Polish National Center for Research and Development). The development team utilizes the Gerrit code review application, and it has already tried out the prototype Android tool with positive feedback. This deployment will allow us to compare both tools (Gerrit vs Android tool), which may produce interesting results.

For this purpose, the Android tool must be further enhanced. The most demanded features are:

- ability to open many files at once,
- further tweaks for application usability (ability to change font size, color and line wrapping behavior),
- *diff* feature allowing to display previous and current version of the code,
- support for version control systems,
- support for communication with existing code review tools for desktop.

References

[1] Cohen J.: *Best Kept Secrets of Peer Code Review*. Printing Systems, 2006, ISBN 9781599160672, `http://books.google.pl/books?id=b9ywHanWN5kC`.

[2] Fagan M.E.: Design and code inspections to reduce errors in program development. *IBM Systems Journal*, vol. 15(3), pp. 182–211, 1976, `http://goo.gl/BDJREG`.

[3] Fagan M.E.: Advances in software inspections. In: *Pioneers and Their Contributions to Software Engineering*, pp. 335–360, Springer, 2001.

[4] Freimut B., Briand L.C., Vollei F.: Determining Inspection Cost-Effectiveness by Combining Project Data and Expert Opinion. *IEEE Transactions on Software Engineering*, vol. 31(12), pp. 1074–1092, 2005, ISSN 0098-5589.

[5] Loeliger J., McCullough M.: *Version Control with Git: Powerful tools and techniques for collaborative software development.* O'Reilly Media, 2012.

[6] Martin R.C.: *Clean Code, A Handbook of Agile Software Craftsmanship.* Pearson Education, Inc., 2009.

[7] Mika V. Mäntylä C.L.: What Types of Defects Are Really Discovered in Code Reviews? *IEEE Transactions on Software Engineering*, vol. 35(3), p. 5, 2009.

[8] Milanesio L.: *Learning Gerrit Code Review.* Packt Publishing, 2013.

[9] Radice R.: *High Quality Low Cost Software Inspections.* Paradoxicon Publishing, 2004, ISBN 9780964591318, `http://books.google.pl/books?id=7HwBAAAACAAJ`.

[10] Wells L.: 9 Reasons to Review Code. 2010, `http://blog.smartbear.com/software-quality/9-reasons-to-review-code/`.

Appendix 1 – Source code reviewed during experiment

```java
package pl.fracz.mcr.source;

/*
 2013-10-23, fracz, first implementation
 2013-10-30, fracz, added syntax highlighting
 2014-02-26, fracz, added ability to add voice comment
 */

import android.annotation.SuppressLint;
import android.content.Context;
import android.graphics.Color;
import android.graphics.Typeface;
import android.text.Html;
import android.widget.LinearLayout;
import android.widget.TextView;

import java.io.File;
import java.io.Serializable;
import java.util.List;

import pl.fracz.mcr.comment.Comment;
import pl.fracz.mcr.comment.CommentNotAddedException;
import pl.fracz.mcr.comment.TextComment;
import pl.fracz.mcr.comment.VoiceComment;

/**
 * View that represents one line of code.
 */
@SuppressLint("ViewConstructor")
public class Line extends LinearLayout implements Serializable {
    private static final long serialVersionUID = 3076583280108678995L;
    private static final int TWO = 2;

    private final int _lineNumber;

    private final String _lineOfCode;

    // holds the line number
    private final TextView lineNumberView;

    private final TextView lineContent;
```

```java
    private final SourceFile sourceFile;

    private List<Comment> comments;

    public Line(Context context, SourceFile sourceFile, int lineNumber,
                String lineOfCode, boolean syntaxColor) {
        super(context);
        this.sourceFile = sourceFile;
        this._lineNumber = lineNumber;
        this._lineOfCode = lineOfCode;
        setOrientation(LinearLayout.HORIZONTAL);

        lineNumberView = new TextView(getContext());
        lineNumberView.setText(String.format("%d.", lineNumber););
        lineNumberView.setSingleLine();
        lineNumberView.setWidth(30);
        addView(lineNumberView);

        TextView lineContent = new TextView(getContext());
        addLineContent(syntaxColor);

        this.comments = sourceFile.getComments().getComments(this);
    }

    public int get() {
        return _lineNumber;
    }

    /**
     * Adds a text comment.
     *
     * @param comment
     * @throws CommentNotAddedException
     */
    public void addTextComment(String comment) throws CommentNotAddedException {
        sourceFile.getComments().addComment(this, new TextComment(comment));
        this.comments = sourceFile.getComments().getComments(this);
        if (comments.size() > 0) {
            lineNumberView.setBackgroundColor(Color.parseColor("#008000"));
        }
    }

    /**
     * Adds a voice comment.
     *
     * @param recordedFile
     * @throws CommentNotAddedException
     */
    public void createVoiceComment(File recodedFile) throws CommentNotAddedException {
        sourceFile.getComments().addComment(this, new VoiceComment(recodedFile));
        this.comments = sourceFile.getComments().getComments(this);
        if (comments.size() > 0) {
            lineNumberView.setBackgroundColor(Color.parseColor("#008000"));
        }
    }

//    public void addVideoComment(File videoFile) throws CommentNotAddedException {
//
//    }

    private void addLineContent(boolean syntaxColor){
        if (!syntaxColor || !SyntaxHighlighter.canBeHighlighted(syntaxColor))
            lineContent.setText(Html.fromHtml(lineOfCode));
```

```
    else
        lineContent.setText(SyntaxHighlighter.highlight(Html. fromHtml(lineOfCode)));
    lineContent.setTypeface(Typeface.MONOSPACE);
    addView(lineContent);
  }

  public List<Comment> getComments() {
    return this.comments;
  }

  public boolean hasConversation(){
    sourceFile.markConversation(this);
    return getComments().size() > TWO;
  }
}
```

Listing 1. Source code reviewed during experiment

Appendix 2 – Full list of known code smells in the reviewed code

Smell codes from [6].

#	Lines	Smell code	Description / expected remarks
1.	3–7	C1	Useless comment, such information should be stored in VCS
2.	21–24	J1	*Wildcard* import of `pl.fracz.mcr.comment.*` would shorten the list of imports
3.	29	G4	Add constructor instead of warning suppression
4.	31	G4	Overridden safety (need to update `serialVersionUID` on every change)
5.	32, 118	G25	`TWO` constant has no logical meaning
6.	34, 36	N6	Useless prefixes (encodings)
7.	38	C2	Obsolete comment
8.	47–48	F1	Too many arguments
9.	47–64	G30	Constructor is too long; it does more than one thing
10.	48	F3, G15	Flag (`boolean`) argument should be replaced by another method
11.	56	–	Syntax error (extra ";")
12.	58	G25	Magic value – a number
13.	61	–	Semantic error; value is stored in local variable instead of class field; as a result, a `NullPointerException` is being thrown when class is instantiated
14.	62	G10	Hidden coupling; method execution relies on previous one, but it is not ensured by its arguments
15.	62	G31	Vertical distance; method is declared far away from its usage
16.	64, 78, 79, 92, 93	G14	"Train wrecks", feature envy, Demeter Law violation

#	Lines	Smell code	Description / expected remarks
17.	64, 79–82, 93–96	G5	Code duplication, DRY
18.	67	N1, N4	Ambiguous method name
19.	71–76, 85–90	C3	Useless comment
20.	77, 91	G11	Inconsistency of method names that perform similar tasks
21.	80, 94	–	*Reinventing the wheel* – use `comments.isEmpty()` instead of comparing its size to zero
22.	81, 95	G25	Magic value – string
23.	91	–	Typo in argument name – `recodedFile` instead of `recordedFile`
24.	99–101	C5	Commented out, not used source code
25.	103, 116	G24	Inconsistent code formatting; lack of space before opening curly brace
26.	104	G28, G29	Overcomplicated, negative boolean expression
27.	104–107	G24	Lack of curly braces around `if...else` blocks
28.	105, 107	G5, G19	Code duplication; result of `Html.fromHtml(lineOfCode)` might be saved to explanation variable
29.	113	–	Method should return defensive copy of a list
30.	117	G20, G30, N7	Method has side effects; it is doing more than its name suggests

Affiliations

Wojciech Frącz

    AGH University of Science and Technology, Institute of Computer Science, Kraków, Poland, `fracz@agh.edu.pl`

Jacek Dajda

    AGH University of Science and Technology, Institute of Computer Science, Kraków, Poland, `dajda@agh.edu.pl`