On the Need for a New Generation of Code Review Tools

Tobias Baum and Kurt Schneider

FG Software Engineering
Leibniz Universität Hannover
Hannover, Germany
{firstname.lastname}@inf.uni-hannover.de

Abstract. Tool support for change-based code review is gaining widespread acceptance in the industry. This indicates that the current generation of tools is well-aligned to current code review practices. Nevertheless, we believe that further improvements in code review tooling can lead to increased review efficiency and effectiveness. In this paper, we combine results from a qualitative study and results from the literature to substantiate this claim. We derive promising improvement areas and provide an overview of existing research in these areas. A common attribute of these improvements is that they trade flexibility for reviewer support. As flexibility is one of the main characteristics of the current generation of code review tools in Hedberg's classification of review tool generations, we regard these coming tools as part of a new generation of code review tools.

Keywords: Code Reviews, Code Inspections and Walkthroughs, Tool Support

1 Introduction

Code review is a well-established method of software quality assurance. In recent years, change-based review has become the dominant style of code review in industry [5,22]. Its main characteristics are the use of code changes performed in a unit of work, e.g. a user story, to determine the scope of the review, and the replacement of management intervention through conventions or rules for many decisions [5]. Change-based review is supported by tools, in some cases specialized code review tools, in other cases general-purpose tools like "diff". Looking at their widespread adoption, these tools seem to address the current needs of the industry. Nevertheless, we believe there is still room for improvement, not least because these tools do not fully incorporate existing research results. The purpose of this article is to derive and collect promising ideas to improve code review effectiveness and efficiency through code review tools in the context of industrial software development. This is done from the point of view of researchers developing these tools and based on results published in the literature and on interviews with software development professionals. This article can be used to guide and direct future research as well as tool development efforts.

2 Methodology

The style of this article is largely deductive. We extract well-founded hypotheses from existing research and combine them to derive and evaluate improvement opportunities for industrial code review. While there are notable differences between classic Fagan Inspection and modern change-based code review, many important aspects are similar [5,22]. We therefore also include results from research on classic inspections as evidence, as long as we believe them to be applicable. Further experiments could be conducted to ascertain these assumptions.

We recently performed a study based on semi-structured interviews with 24 software professionals [5,6]. To some extent, these interviews concerned the way in which reviewers work and which problems they perceive. These interviews form the second pillar of our argumentation, in addition to the literature. Due to space limitations, we cannot provide methodological details on these interviews and have to refer to our related articles [5,6]. We cite many statements from the interviews as examples for certain points. The subscripts at these citations denote the interviewee ID from [6].

3 What Do We Know about Code Review?

A lot of research has been done on code reviews and inspections, and still many questions could not be answered conclusively. But some results are relatively well supported and a subset of these will form the foundation of our discussion:

The first such result concerns which factors have a major and which only a minor influence on the effectiveness and efficiency of reviews. When analyzing experimental data, Porter et al. "found that [reviewers, authors, and code units] were responsible for much more variation in defect detection than was process structure", and they "conclude that better defect detection techniques, not better process structures, are the key to improving inspection effectiveness." [20] A similar conclusion is reached by Sauer et al., who identify "individuals' task expertise as the primary driver of review performance" based on theoretical considerations. Correlations between the (inspection) expertise of the reviewer and the number of found defects have also been reported by Rigby [21] and by Biffl and Halling [8], just to name a few. We conclude that the major factors influencing code review effectiveness and efficiency are the reviewer, its relation to the artifact under review and the way in which it performs the checking.

The second important result is about the role of understanding the artifact under review. In their study based on interviews with developers at Microsoft, Bacchelli and Bird found that "[m]any interviewees eventually acknowledged that understanding is their main challenge when doing code reviews" [2], which confirmed earlier results from Tao et al. [23]. Further support for a positive correlation between code understanding and review effectiveness comes from experiments by Dunsmore, Roper and Wood [12]. Our interview results fully support these findings, e.g.: "I have to understand what the other developer thought at that time. And for that you look very closely at the code, and then things that should or could be done better somehow come up automatically" [3].

4 The Problem of Large Changes

In our interviews, we asked about problems hampering review effectiveness. One of the most common themes was the difficulty to understand and review large changesets: "Smaller commits are generally not a problem. But these monster commits are always . . . not liked very much by the reviewers." [5] "What sometimes impedes me is when the ticket is just too big." [7] "When you have such a big pile to review the motivation is not very high and you probably don't approach the review with the needed quality in mind." [12]

The conclusion that large changesets are problematic can also be derived from other research results: There is evidence that the review effectiveness greatly decreases when the review rate (checked lines of code/time for checking) is outside the optimal interval (see e.g. [15]). There is also evidence that concentration and therefore review effectiveness fades after some time of reviewing. Combining these values leads to an upper limit on the maximal size of an artifact that can be reviewed effectively in a single session.

Given the problems with the review of large changes, many teams resort to the frequent review of small changes [21]. Up to a certain point, this is a good thing to do, but there are also arguments in favor of larger changes and reviews: The change under review should be self-contained, it should fulfill certain quality criteria before central check-in (at least to be compilable) and reviewing very small changes can lead to high overhead and duplicate work [23]. So instead of forcing every change to be very small, we argue to make the review of larger changes more effective and let changes stay at their "smallest natural size".
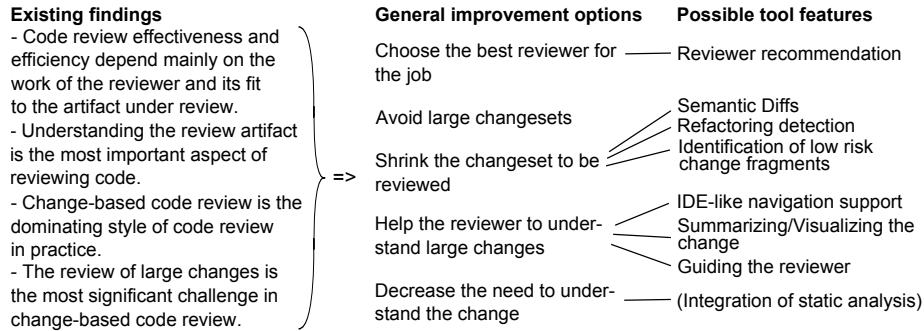
5 Tool Support to the Rescue

We substantiated in the previous sections that to increase the effectiveness and efficiency of code reviews, we should focus on the reviewer and how to help her understand large code changes better. We believe that improved tool support provides a lot of opportunities in this regard, and will give examples in the following subsections. Additionally, an overview of our argumentation is shown in Fig. 1. The subsections correspond to the most important influencing factors, deduced from the results mentioned so far:

- Choose the best reviewer for the job (Sect. 5.1)
- Shrink the size of the changeset that has to be reviewed (Sect. 5.2)
- Help the reviewer to understand large changesets (Sect. 5.3)
- Decrease the need to understand the change (Sect. 5.4)

5.1 Reviewer Recommendation

In recent years, there have been a number of studies on "reviewer recommendation", i.e. on finding the best reviewer(s) for a given change (e.g. [3, 26]). While this promises a large effect in theory, there are several problems reducing the

Existing findings
- Code review effectiveness and efficiency depend mainly on the work of the reviewer and its fit to the artifact under review.
- Understanding the review artifact is the most important aspect of reviewing code.
- Change-based code review is the dominating style of code review in practice.
- The review of large changes is the most significant challenge in change-based code review.

=>

General improvement options

Choose the best reviewer for the job

Avoid large changesets

Shrink the changeset to be reviewed

Help the reviewer to understand large changes

Decrease the need to understand the change

Possible tool features

Reviewer recommendation

Semantic Diffs
Refactoring detection
Identification of low risk change fragments

IDE-like navigation support
Summarizing/Visualizing the change
Guiding the reviewer

(Integration of static analysis)

Fig. 1. Overview of argumentation and tool features

benefit, especially in smaller teams. The most obvious is that in a small team, it is often fairly easy to see who is a good reviewer for a change, so that computer support does not provide large benefits. Additionally, always choosing the best reviewer can lead to a high review load for experienced developers, and a high workload has a negative impact on review quality [7]. Therefore reviewer recommendation has to move from determining local optima for every single review to a more global optimization of reviewer assignment.

5.2 Reducing Changeset Size

Given large changesets with singular changes of varying risk, reviewers try to manually pick the relevant subset. This is seen as a hard and error-prone task: "After some time you get a feeling which files are relevant and which are not, but it's hard to filter them out. And, well, when I don't look at them there might be some change in there that was relevant, anyway. That's problematic." $_8$

An important special case is systematic changes, especially rename and move refactorings. This special case is much better studied than the general case, for example with works from Tao and Kim [24], Thangthumachit, Hayashi and Saeki [25] and Ge [14]. Support for the general case has been studied by Zhang et al. [27] and by Kawrykow and Robillard [18]. Further research could provide a better foundation to decide which changes are low-risk, and it could look into the distinction between change fragments that are error-prone and need to be checked in detail and change fragments that only need to be read to help understanding. Nevertheless, much could already be gained by bringing the promising existing results into wider use.

5.3 Support for Understanding the Change

A theme that occurred throughout our interviews is that large changes are best reviewed with the search and hyperlinking support of an IDE (e.g. "I think reviewing code purely in 'Crucible' only works for trivialities. Because naturally many features are missing that you have in an IDE." $_2$). This improvement has

already made its way into some widely used review tools, either by making IDE-like support available in a browser (e.g. "Upsource") or by making the review tool available as an IDE plugin (e.g. "AgileReview" or "EGerrit").

Many of our interviewees try to get a high-level understanding of the change at the start of the review ("at first an overview because otherwise the problem is that you loose sight of the interrelation of the changes" $_{10}$). The current support for this activity is very limited, consisting mainly of the overview of the commit messages of the singular commits belonging to the change. There is relatively little research on visualizing and summarizing code changes for better understanding: McNair, German and Weber-Jahnke propose an approach to visualize change-sets [19], as do Gomez, Ducasse and D'Hondt [16]. In addition, several textual summarization techniques have been proposed (e.g. [9]). A related technique that can help to summarize the contents of a change is "change untangling" [4, 11, 24]. We believe that more research on these topics is needed to make change visualization effectively usable by reviewers.

After having an overview of the changes, the reviewer needs to step through the change's details in some order. Many reviewers try to find an order that helps their understanding, but often fall back to the order presented by their review tool: "The problem is you sometimes get lost and don't find a good starting point." $_{10}$ "If you don't have that, you just steps through the files in the commit one after another . . . " $_{10}$. A similar finding resulted from a study by Dunsmore, Roper and Wood where participants suggested "ordering of code" to improve inspections [13]. Guiding the reviewer as proposed here shares some similarities with the reading techniques studied intensively for inspections [1, 10]. The main difference is that these reading techniques try to change the way the reviewer works, while the proposed guiding moves some cognitive load from the human reviewer to the tool. In addition, most reading techniques proposed so far are not intended to be used with changesets, so that research opportunities abound in this area.

5.4 Decrease the Need for Code Understanding

From a theoretical point of view, reducing the need to understand the code is another possibility to solve the stated problem. Essentially this is a question of efficiency: Is in-depth code review the most efficient way to find a certain defect type or are there more efficient ways, e.g. static code analysis or testing? As long as there are practically relevant defect types for which in-depth code review is most efficient, understanding the code will still be needed. And when there will be no such defect types anymore, for example after a breakthrough in static analysis research, code review in its current form will not be needed any longer for defect detection. Therefore, we won't discuss this topic further in this article.

6   A New Generation of Code Review Tools

About a decade ago, Henrik Hedberg proposed a classification of software inspection/review tools into generations [17]. He concluded that the coming fifth

generation should provide flexibility with regard to the supported documents and processes and that they should comprehensively include existing research results. This prediction has come true (with limitations): Current review tools like "Gerrit", "Crucible" or "Collaborator" are flexible and commonly support the review of any kind of text file. In the preceding sections, we derived opportunities to reach a higher level of review effectiveness. For most of them, a reification from the review of changes in text files to the review of changes in source code has to take place. Flexibility is traded for better reviewer support to some degree. This leads us to expect the rise of the sixth generation of code review tools, the generation of "cognitive support review tools".

7 Summary

We collected three findings on code review we regard as well established: (1) Code review effectiveness and efficiency depend to a large degree on the reviewer, its style of work and its fit to the artifact under review. (2) Understanding the review artifact is the most important aspect of reviewing code. (3) The review of large changes is the most significant challenge in change-based code review. Based on these assumptions we derived leverage points to improve review effectiveness and efficiency through tool support. For each of these points, we surveyed existing research and state open research questions. In our own work, we currently look into some of these research questions. We believe there is an abundance of open questions for other researchers to join us in our efforts to lay the foundation for the generation of "cognitive support review tools".

References

1. A. Aurum, H. Petersson, and C. Wohlin. State-of-the-art: software inspections after 25 years. *Software Testing, Verification and Reliability*, 12(3):133–154, 2002.
2. A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proc. 2013 Int. Conf. on Software Engineering*. IEEE, 2013.
3. V. Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *Proc. 2013 Int. Conf. on Software Engineering*. IEEE Press, 2013.
4. M. Barnett, C. Bird, J. Brunet, S. K. Lahiri, and M. Barnett. Helping developers help themselves: Automatic decomposition of code review changesets. In *Proc. 2015 Int. Conf. on Software Engineering. IEEE Press*, 2015.
5. T. Baum, O. Liskin, K. Niklas, and K. Schneider. A faceted classification scheme for change-based industrial code review processes. In *QRS*, 2016. pending publication, pre print at http://tobias-baum.de/rp/classification.pdf.
6. T. Baum, O. Liskin, K. Niklas, and K. Schneider. Factors influencing code review processes in industry. In *FSE*, 2016. pending publication, pre print at http://tobias-baum.de/rp/factors.pdf.
7. O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey. Investigating technical and non-technical factors influencing modern code review. *Empirical Software Engineering*, pages 1–28, 2015.

8. S. Biffl and M. Halling. Investigating the influence of inspector capability factors with four inspection techniques on inspection performance. In *Software Metrics, 2002. Proc. Eighth IEEE Symp. on*, pages 107–117. IEEE, 2002.

9. R. P. Buse and W. R. Weimer. Automatically documenting program changes. In *Proc. IEEE/ACM Int. Conf. on Automated software engineering*. ACM, 2010.

10. C. Denger, M. Ciolkowski, and F. Lanubile. Investigating the active guidance factor in reading techniques for defect detection. In *Empirical Software Engineering, 2004. Proc. Int. Symp. on*, pages 219–228. IEEE, 2004.

11. M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse. Untangling fine-grained code changes. In *Software Analysis, Evolution and Reengineering, 2015 IEEE 22nd Int. Conf. on*, pages 341–350. IEEE, 2015.

12. A. Dunsmore, M. Roper, and M. Wood. The role of comprehension in software inspection. *Journal of Systems and Software*, 52(2):121–129, 2000.

13. A. Dunsmore, M. Roper, and M. Wood. Systematic object-oriented inspection – an empirical study. In *Proc. 23rd Int. Conf. on Software Engineering*. IEEE, 2001.

14. X. Ge. *Improving Tool Support for Software Developers through Refactoring Detection*. PhD thesis, North Carolina State University, 2014.

15. T. Gilb and D. Graham. *Software Inspection*. Addison-Wesley, 1993.

16. V. U. Gómez, S. Ducasse, and T. D'Hondt. Visually characterizing source code changes. *Science of Computer Programming*, 98:376–393, 2015.

17. H. Hedberg. Introducing the next generation of software inspection tools. In *Product Focused Software Process Improvement*, pages 234–247. Springer, 2004.

18. D. Kawrykow and M. P. Robillard. Non-essential changes in version histories. In *Proc. 33rd Int. Conf. on Software Engineering*, pages 351–360. ACM, 2011.

19. A. McNair, D. M. German, and J. Weber-Jahnke. Visualizing software architecture evolution using change-sets. In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conf. on*, pages 130–139. IEEE, 2007.

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

21. P. C. Rigby. *Understanding open source software peer review: Review processes, parameters and statistical models, and underlying behaviours and mechanisms*. PhD thesis, University of Victoria, 2011.

22. P. C. Rigby and C. Bird. Convergent contemporary software peer review practices. In *Proc. 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013.

23. Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim. How do software engineers understand code changes?: an exploratory study in industry. In *Proc. ACM SIGSOFT 20th Int. Symp. on the Foundations of Software Engineering*. ACM, 2012.

24. Y. Tao and S. Kim. Partitioning composite code changes to facilitate code review. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conf. on*, pages 180–190. IEEE, 2015.

25. S. Thangthumachit, S. Hayashi, and M. Saeki. Understanding source code differences by separating refactoring effects. In *Software Engineering Conf. (APSEC), 2011 18th Asia Pacific*, pages 339–347. IEEE, 2011.

26. P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K.-i. Matsumoto. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd Int. Conf. on*, 2015.

27. T. Zhang, M. Song, J. Pinedo, and M. Kim. Interactive code review for systematic changes. In *Proc. 37th IEEE/ACM Int. Conf. on Softw. Engineering. IEEE*, 2015.