Personal Reading: Code Review

Mats Stijlaart - University of Amsterdam, The Netherlands

February 28, 2015

What is Code Review?

Code Review is a part of a development process where software developers inspect code to find defects [3, p. 47]. The company developing this software conducts this process. Not an external party. Companies apply Code Reviews to deduce the amount of defects in their code base in an earlier stage. Code reviews, also called peer reviews, catch an average of 60% of the defects of the reviewed code base [2, p. 136]. Different studies have shown that defects in a later stage of a project tend to cost more money to resolve than in earlier phases [2, p. 135] [1, p. 21]

What kind of Code Review Exist?

Different types of code review exist. For example [5, pp. 23–38]:

- **Formal Inspections** [5, p. 23]
  A Formal Inspection code review is a meeting where developers come together and review a piece of software. For example, the code will be presented on a beamer and the developers will note corrections when they find defects. The found defects can be formalized and applied after possible discussion.
- **Over-the-Shoulder Reviews** [5, p. 26]
  An over-the-Shoulder Review is a review session where a colleague inspects a piece of changed software with the guidance of the programmer that did built it. The developer will talk his colleague through the made changes and feedback can be easily addressed or directly fixed in the code base.
- **E-mail Pass-Around Reviews** [5, p. 30]
  A developer will gather changes on a piece of software for a certain time-frame and will send them around in email to a group of developers. Developers can reply on the email with the defects they have found.
- **Tool-Assisted Reviews** [5, p. 34]
  A tool is employed to assist with the code review. When developers want to perform a code review, they use the tool to invite other developers to perform a review. The tool will support the changed files and the changed content in a convenient manner and the other developers can comment on the changed code within the tool.
- **Pair-Programming** [5, p. 37]
  Pair-Programming is a practice where two developers work on a single task behind a single machine. One programmer will do the actual programmer (the driver), while the other developer (the navigator) will continuously check the code and gives feedback. This feedback is a form of code review, but also contains additional benefits such as learning from each other.

Cohen et al. describe these forms, but I can imagine there are additional forms. For example, on Friday afternoon with the delight of a beer you can join with a view colleagues to review a certain module. This may be an informal alternative of Formal Inspection. Some of these five types are cost-intensive, such as Formal Inspections and E-mail Pass-Around Reviews. The main cost will be addressed in the preparation, multiple

people dedicating their time and collecting the necessary information [5, pp. 23–38]. One statement of Cohen et al. I found a bit odd. They state that E-mail Pass Around Reviews "is the second-most common form of informal code review, and the technique preferred by most open-source projects" [5, p. 30]. I have worked in open-source projects and the main code collaboration tooling such as BitBucket[1], GitHub[2], GitLab[3] and FishEye[4] all have Tool-Assisted Reviews. Cohen et al. published their work in 2006. I believe that last ten years changed a lot in this area.

I have never attended a Formal Inspection but can imagine, just as Cohen et al. address, they are of high costs due to the high number man-hours allocated for the task. Votta et al. showed that around 96% of the defects could be found when the same individuals reviewed the code on their own [8, p. 110]. Due to the unknown, how much money the remaining 4% may cost it may be cost saving when using Tool-Assisted Reviews instead of Formal Inspections.

In my experience in the industry I have only worked with Pair-Programming and Tool-Assisted Reviews and a little with Over-the-Shoulder review. Personally I think that the others are a bit inconvenient. For example, for Formal Inspection you have to gather a group of developers on a certain time. I know my colleagues and none of them every 'has time' and additionally with Formal Inspection you will force a developer to stop his current tasks and switch context on possibly an inconvenient moment. This together with the findings from Votta presented in the previous paragraph may show that Tool-Assisted Reviews are more convenient than Formal Inspections. Tool-Assisted Reviews allow developers to allocate time when it fits their workflow, which I personally find important while switching context (from your code to someone else's) is hard.

The same holds for E-mail Pass-Around Review. There is a workload to gather all changes, send them to colleagues wait for responses and than gather them all together. This seems to be an administrative expensive task and error prone.

I wanted to make a last remark on these types of code review related to Over-the-Shoulder Reviews. I have only encountered these for bug fixes that had to be deployed on a production environment as fast as possible. These fixes are often just a one-line patch. Because a formal review was too time consuming and the context was too critical to just commit the code an extra pair of eyes would validate the patch. In my opinion this is an appropriate form of code reviewing whereas the situation.

Merits and Drawbacks

As described earlier, code reviews have merits and drawbacks. The main advantage is the elimination of defects in an early phase, which gains a decrease in cost as mentioned in *What is Code Review?* and less defects imply less faults in the software. This is directly contradicted with the fact that people have to spend time on the code review. It is unknown if these outweigh each other or if is an umbrella for the other.

The researched literature omits some of the advantages that I actually find pretty obvious. The encountered information mainly focuses on the price tag and the design of code reviews. Additional benefits that I see are:

- For low fault-tolerant systems it is additional manner to find defects.

[1] https://bitbucket.org/
[2] https://github.com/
[3] https://about.gitlab.com/
[4] https://www.atlassian.com/software/fisheye

- When reviewing somebody else his code, there is automatically co-ownership of the code. At least two developers in the organization know about the change. This can benefit the organization when the first developer gets sick or even leaves the company. Especially in smaller teams this may be beneficial where the impact of a missing employee may be more significant.
- Code reviewing will help with code-style. When developers write their code in different styles, a code review is an optimal process to discuss these divergent ideas. One can argue that these style issues are covered in a style-guide, but naming conventions such as *findEntity* or *getEntity* for function names are sometimes not covered. Code reviews can also help to expand the style-guide.
- It is a learning and control mechanism for junior programmers, interns or new team members. Inexperienced programmers can learn from more experienced programmers by reading their code. Also, the experienced programmers can check if the inexperienced developers do not add any invalid code. This may be beneficial in a complex domain. For example, financial regulations.

I think these additional benefits may prevail the choice to start using code reviews.

When should you use Code Reviews?

It is not clear if code reviews out-perform other techniques like functional testing. However, it is shown that it is likely that different types of defects are found with the different types of code checking [7]. This uncertainty may be removed in the industry by just applying code reviews and measure what happens with the velocity of a team and the amount of defects. Especially if one of the additional advantages, mentioned in the previous paragraph, applies to the organization I would advice to try it.

How to apply Code Reviews?

The writer of the *Making Software* book presents that when somebody reviews a piece of code, the number of lines should not exceed 400 and reviews should not take longer than 60 minutes [4, pp. 330-332] [5, p. 81] [6, pp. 470–471]. Otherwise this results in a significant performance loss. Again, whether this performance loss outweighs the cost of finding defects later in the process is unknown.

From my personal experience I can verify the maximum of 400 lines. I encountered pull requests that covered over 1000 lines of code. One way or another these always impacted my motivation negatively. For example, pulling me out of my daily activities for too long. The 60-minute boundary I encountered less often. The environment where I performed my code reviews were usually supported by a good number of automated tests, which allowed me to focus more on the structure of the code instead of the actual functioning (this would be checked by the build server). I would estimate my review rate in my daily job on 100 lines/5 minutes. This makes me verge towards the advice to accompany code reviews with functional tests to decrease the workload.

Conclusion

Code reviews will help developers to decrease the number of defects in a code base. The main consideration to use code reviews is the gain of quality and decrease of cost in a later phase against the extra allocation of people (and thus costs) to perform the code reviews. The actual costs are unknown, but as described in *Merits and Drawbacks*

there are additional advantages that may help an organization to decide to use code reviews. In my opinion I would directly start use code reviews in an quality demanding environment.

References

[1] Kent Beck. *Extreme programming explained: embrace change*. addison-wesley professional, 2000.

[2] Barry Boehm and Victor R. Basili. "Software Defect Reduction Top 10 List". In: *Computer* 34.1 (2001), pp. 135–137. ISSN: 0018-9162.

[3] Marcus Ciolkowski, Oliver Laitenberger, and Stefan Biffl. "Software Reviews: The State of the Practice". In: *IEEE Software* 20.6 (2003), pp. 46–51. ISSN: 0740-7459. DOI: http://doi.ieeecomputersociety.org/10.1109/MS.2003.1241366.

[4] Jason Cohen. "Modern Code Review". In: *Making Software: What Really Works, and Why We Believe It*. 1st ed. O'Reilly, 2011. Chap. 18, pp. 329–338.

[5] Jason Cohen et al. *Best kept secrets of peer code review*. Smart Bear, 2006.

[6] Alastair Dunsmore, Marc Roper, and Murray Wood. "Object-oriented Inspection in the Face of Delocalisation". In: *Proceedings of the 22Nd International Conference on Software Engineering*. ICSE '00. Limerick, Ireland: ACM, 2000, pp. 467–476. ISBN: 1-58113-206-9. DOI: 10.1145/337180.337343. URL: http://doi.acm.org/10.1145/337180.337343.

[7] Natalia Juristo and Sira Vegas. "Empirical Methods and Studies in Software Engineering: Experiences from ESERNET". In: ed. by Reidar Conradi and Alf Inge Wang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. Chap. Functional Testing, Structural Testing and Code Reading: What Fault Type Do They Each Detect?, pp. 208–232. ISBN: 978-3-540-45143-3. DOI: 10.1007/978-3-540-45143-3_12. URL: http://dx.doi.org/10.1007/978-3-540-45143-3_12.

[8] Lawrence G. Votta Jr. "Does Every Inspection Need a Meeting?" In: *Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering*. SIGSOFT '93. ACM, 1993, pp. 107–114. ISBN: 0-89791-625-5. DOI: 10.1145/256428.167070. URL: http://doi.acm.org/10.1145/256428.167070.