How to do effective code reviews

26 June 2017 | **Insight Zuhlke, Software Engineering** | **Oliver Zihler**
Reading time: 7 minutes

Frank Sons (code-quality.de) held an interesting talk at Jax 2017 about how to make code reviews more effective which I summarise and elaborate on in the following.

Status quo of code reviews
We all know it from our own experiences and we certainly do not need to argue about it: We (meaning, I) write the best code, and the code of all other developers is utter nonsense (I will not write complete BS; that's just unprofessional, although true). There is usually only one occasion after which we might reconsider our opinion: When our fellow dev colleagues review (and most likely criticise) our code. Only then we learn that criticism of own code hurts. In summary, many of us have some kind of *ego problem*, which very likely hinders us from doing more effective code reviews. However, as Jeff Atwood said: "Peer code reviews are the single biggest thing you can do to improve your code". Nevertheless, why do we actually do code reviews?

Reason for Code Reviewing	**Rank**
Code improvement	1
Find defects	2
Increase knowledge transfer	3
Find alternative solutions	4
Improve the development process	5
Avoid breaking builds	6
Build team awareness	7
Shared code ownership	8
Team assessment	9

Table 1. Developers' ranking of perceived reasons for reviewing code.

Asking developers this question, studies found that the main perceived goals are to improve code quality, uncover defects, increase knowledge transfer, discuss alternative solutions, etc. (see Table 1). However, an empirical study showed that around 50% of the actual improvements have nothing to do with fixing bugs. Instead, they relate to improving the code's documentation, adjusting code arrangement, discussing alternative solutions and polishing up the appearance of the code (like removing blank lines, adapting indentions, etc.). Only a mere 15% contribute to the prevention of defects, indicated by table 2 (see goo.gl/M5srAV and cited literature for further information). Thus, developers' idea of what they achieve during code reviews does not necessarily match the eventual outcome of a code review. This also has the consequence that developers do not actually know if their

code reviews are effective. Its effectiveness heavily depends on experience, timing, and motivation of the reviewing developers.

Comments category	Types of issues included	Freq.
Documentation	Comments, naming, style	22.3%
Organisation of code	Modularity, location of artefacts, new class, duplicate code, size of methods	15.9%
Solution approach	Alternate algorithm or data structure	8.5%
Validation	Lack of or improper validation	7,2%
Visual representation	Beautification, indention, blank lines	6,4%
False positive	Not a real issue	4.6%
Defect	Incorrect implementation or missing functionality	2.6%
Logical	Control flow or logic issues	2.3%
Support	Configuration support systems or libraries	2.0%
Interface	Interactions with other components	1.5%
Resources	Resource initialisation, manipulation, and release	1.3%
Timing	Thread synchronisation, races	0.3%
Other		25.1%

Table 2. Classification of code review comments show that around 50% of the comments are attributed to documentation, organization, solution approach, and visual representation of code (indicated by red highlighting), whereas only 15% of the comments (blue) aim at preventing code errors.

More effective code reviews

In summary, generalised and exaggerated, one can say developers can be a bunch of ego-driven know-alls whose perception of the purpose of code reviews differs quite a bit from what they eventually end up doing. The question thus is if there are simple, applicable measures to improve the quality of code reviews.

Leave your ego at the door

To quote Jeff Atwood again, a first step towards better code reviews is to "*attempt to be awesome in public and embrace to suck*". Obviously, developers need to leave their ego at the door. They should embrace the feedback they get on their code to learn from it and improve themselves (although they do not necessarily think they need it). Developers have to abandon the thought of "their" code: it is the team's responsibility, so share the knowledge and be open for alternatives. All members of the team should become addicted to getting feedback on their code to improve and develop a common mind set for code quality.

Compile a guideline, checklist and solution templates

Next, the intentions for doing code reviews need to be clear to every developer. If the goal is to find defects or alternative solutions a solid understanding of the code by the reviewer is an indispensable prerequisite. In many cases, a guideline can help developers keep their intentions on why they do code reviews in line. Every team should create an individual *guideline* to fit their specific understanding of code quality (incorporating any mandatory business requirements, of course).

The guideline is a living document, meaning reflection on its effectiveness is of uttermost importance. If some points become obsolete, consider adapting or removing them completely. At the same time, new circumstances may require the addition of new paragraphs to the guideline. A *checklist* used during every code review can help enforcing the guideline. Of course, the checklist should be small enough to actually be applied during code reviews so that it doesn't become a pages long dust catcher.

Additionally, the structure of code reviews should allow deeper discussion of findings *after* the actual review. This allows busy developers an insight into the code, too, without stealing too much of their time. Furthermore, not everything needs to be fixed instantly. It makes sense to write all larger findings down and fix them later, possibly already scheduling a follow-up review. To speed up such code improvements, a compilation of *solution templates* for commonly encountered problems allows a team to operate more independently of the current (and often changing) team constellation.

Helpful tools

The right methods at the right time can facilitate code reviews further. *Pull requests* allow for remote code reviews (but can also get messy if they are overused). In case of *pair programming*, code reviews may not even be needed. *Code review meetings* can be the most beneficial way for sharing knowledge but require the participating people to be willing to pose questions (and some 1-2 hours of their time). A good method to evaluate a code's comprehensibility is to have somebody explain it who hasn't seen it before. It demonstrates the developers under review also the difficulty of writing understandable code. Many developers claim that documentation is not required; the code is its own documentation. However, how quickly do we introduce a hack to fix some unforeseen exceptional case? If this piece of code is not documented, once the development team changes, the knowledge about why it is there is lost, and the code has become instant legacy.

Summary

To summarise, what are the simple steps to make a code review more effective?

1. Leave your ego at the door. Try to enjoy criticism. Lead by example and initially, only review code of developers that are able to handle criticism. Others will likely follow once they see the benefits.
2. Have a guideline and a checklist to follow. Have solution templates for common problems.
3. Follow up on results: Bigger issues may need a follow-up review.
4. Use the right tools at the right time.
   Again, as Jeff Attwood said: "P*eer code reviews are the single biggest thing you can do to improve your code!*"

   What do you think about code reviews? Do you do code reviews in your team and if so, how? Or do you think code reviews are just a waste of time? Leave a comment down below, please.