Two Kinds of Code Review

Jan 3, 2021

I've read a book about management and it helped me to solve a long-standing personal conundrum about the code review process. The book is "High Output Management". Naturally, I recommend it (and read this "review" as well: https://apenwarr.ca/log/20190926).

One of the smaller ideas of the book is that of the managerial meddling. If my manager micro-manages me and always tells me what to do, I'll grow accustomed to that and won't be able to contribute without close supervision. This is a facet of a more general **T**ask-**R**elevant **M**aturity framework. Irrespective of the overall level of seniority, a person has some expertise level for each *specific* task. The optimal quantity and quality of supervisor's involvement depends on this level (TRM). When TRM grows, the management style should go from structured control to supervision to nudges and consultations. I don't need a ton of support when writing Rust, I can benefit a lot from a thorough review when coding in Julia and I certainly require hand-holding when attempting to write Spanish! But the overarching goal is to improve my TRM, as that directly improves my productivity and frees up my supervisor's time. The problem with meddling is not excessive control (it might be appropriate in low-TRM situations), it is that meddling removes the motivation to learn to take the wheel yourself.

Now, how on earth all this managerial gibberish relates to the pull request review? I now believe that there are two largely orthogonal (and even conflicting) goals to a review process.

One goal of a review process is good *code*. The review ensures that each change improves the overall quality of a code base. Without continuous betterment any code under change reverts to the default architecture: a ball of goo.

Another goal of a review is good *coders*. The review is a perfect mentorship opportunity, it is a way to increase contributor's TRM. This is vital for community-driven open-source projects.

I personally always felt that the review process I use falls quite short of the proper level of quality. Which didn't really square with me bootstrapping a couple of successful open source projects. Now I think that I just happen to optimize for the people's aspect of the review

process, while most guides (with a notable exception of Optimistic Merging) focus on code aspects.

Now, (let me stress this point), I do not claim that the second goal is inherently better (though it sounds nicer). It's just that in the context of both IntelliJ Rust and rust-analyzer (green-field projects with massive scope, big uncertainties and limited payed-for hours) growing the community of contributors and maintainers was more important than maintaining perfectly clean code.

Reviews for quality are hard and time consuming. I personally can't really review the code looking at the diff, I can give only superficial comments. To understand the code, most of the time I need to fetch it locally and to try to implement the change myself in a different way. To make a meaningful suggestion, I need to implement and run it on my machine (and the first two attempts won't fly). Hence, a proper review for me takes roughly the same time as the implementation itself. Taking into account the fact that there are many more contributors than maintainers, this is an instant game over for reviews for quality.

Luckily, folks submitting PRs generally have medium/high TRM. They were able to introduce themselves to the codebase, find an issue to work on and come up with a working code without me! So, instead of scrutinizing away every last bit of diff's imperfection, my goal is to promote the contributor to an autonomous maintainer status. This is mostly just a matter of trust. I don't read every line of code, as I trust the author of the PR to handle ifs and whiles well enough (this is the major time saver). I trust that people address my comments and let them merge their own PRs (`bors d+`). I trust that people can review other's code, and share commit access (`r+`) liberally.

> Note that explicit calls for contribution such as "good first issue" labels tend to attract less experienced contributors. When applying this label, make sure that you are ready to closely mentor the PR.

What new contributors don't have and what I do talk about in reviews is the understanding of project-specific architecture and values. These are best demonstrated on specific issues with the diff. But the focus isn't the improvement of a specific change, the focus is teaching the author of (hopefully) subsequent changes. I liberally digress into discussing general code philosophy issues. As disseminating this knowledge 1-1 is not very efficient, I also try to document it. Rather than writing a PR comment, I put the text into architecture.md or style.md and link that instead. I also try to do only a small fixed number of review rounds. Roughly, the PR is merged after two round-trips, not when there's nothing left to improve.

All this definitely produces warm fuzzy feelings, but what about code quality? Gating PRs on quality is one, but not the only one, way to maintain clean code. The approach I use instead is continuous reafactoring / asynchronous reviews. One of the (documented) values in rust-analyzer is that anyone is allowed and encouraged to refactor all the code, old and new.

Instead of blocking the PR, I merge it and then refactor the code in a follow-up (ccing the original author), when I touch this area next time. This gives me a much better context than a diff view, as I can edit the code in-place and run the tests. I also don't waste time transforming the change I have in mind to a PR comment (the motivation bits go directly into comment/commit message). It's also easy to do unrelated drive-by fixes!

I wish this asynchronous review workflow was better supported by tools. By default, changes are merged by the author, but the PR also goes to a review queue. Later, the reviewer looks at the merged code in the main branch. Any suggestions are submitted as a new PR, with the original author set as a reviewer. (The in-editor reviewing reminds me iron workflow.)

For conclusion, let me reference another book. I like item 32 from "C++ Coding Standards": be clear what kind of class you're writing. A value type is not an interface is not a base class. All three are classes, but each needs a unique set of rules.

When doing/receiving a code review, understand the context and purpose. If this is a homework assignment, you want to share knowledge. In a critical crypto library, you need perfect code. And for a young open source project, you aim to get a co-maintainer!