Ways to improve your team's code quality

January 24, 2020 - 5 minutes read - 1050 words

There’s an argument that engineers should care about code quality. Teams and companies make specific and targeted efforts to keep the quality of their codebases high. The existence of activities like “spring cleaning”, “test Fridays”, “Fixit week” and others assert the importance of code maintenance, giving engineers the breathing room to fix complex, hairy issues that take more than a day or two of time and focus to solve.

Some engineers (I do) will talk about the problem of “poor quality code”, “bad code”, or “spaghetti code” as if the code wasn’t written by engineers just like ourselves, but the bottom line is, every engineer can likely find a piece of code they wrote that they aren’t proud of today. We’ve since gained experience that allows us to view the contribution with deepened context and in a different light. As more proficient engineers, rather than just seeing a problem and pointing it out, we can do something productive to address the reason the poor code was written in the first place.

I’ll propose a few strategies I’ve developed to try and turn an unavoidable part of life as an engineer into an opportunity for us to help each other continue to learn and get better at the work we do.

When you find a file or function that is the perfect combination of unreadable and poorly named, misusing design patterns and languages features, violating abstractions and code base conventions:

Identify if the pattern exists elsewhere.

The same snippets of code get used and re-used all over the place across codebases and documentation. Copying code is great and fast, especially when the code is good (maybe you should have better abstractions but that is a different conversation). When the code is bad, copying the code can get the job done, but proliferates bad patterns and code entropy increases. If you identify the source of the bad pattern, you can correct/improve it at its source then author changes to help others from doing the same in the future. If you don't make these fixes, the pattern becomes malignant, pervading the codebase in which it originated but also showing in other codebases as less experienced contributors continue following the norms to which they've become accustomed. As an experienced member of your team, you can easily scale your impact by ensuring less experienced contributors the copy patterns you *want* to see in your codebases rather than just whatever they're used to doing.

Consider if the codebase uses patterns that encourage the contributions you want.

Chances are, if you're not happy with the state of your codebase today, you won't be happy with how new contributes look either. Does the code that already exists in the codebase look like new code you'd want added? Do you do a good job of creating focused modules and adhering to separation of concerns? Do your unit tests properly mock out the external layers and packages you're not testing? Figure out what your gold standard looks like and point contributors to that example of what their code should look like, while acknowledging that you're working towards a state where more things look like that gold standard. This is easier said than done – there is a bias to maintain the status quo, even when better alternatives are known. Given this bias, it's helpful to be proactive, telling new contributors something like:

> It would be great if you could follow the pattern we use in the `users` gateway even though it's different from most of the other ones. That is our north star for how new contributions should look.

Additionally, work with your team to carve out time for code maintenance and refactoring so that in the future, less explanation is needed for new contributors.

Pretend that you were the one who wrote the code.

Sometimes you don't have to pretend – you read code, think "hmm, this is not the way I'd do this", and it turns out, you `git blame` and realize you literally wrote it yourself, months or years earlier. Between when you wrote this code and later admonished yourself for the bad code you wrote, you've learned some things that have made you believe you could have done better. Maybe you've come to understand the codebase better, become more familiar with effective abstractions or came up with a design pattern that better fits the use case. Maybe you're just not in as much of a rush this time as your were last time. Regardless of whether you wrote the code or not, at some point there was learning which leads to the thought "I think I know a way to do it better."

Capture these ideas and engage with the author(s) to discuss why they chose not to do it the way you would have. Maybe they'll learn something or maybe you will, and next time, consider leaving a comment or doc on why you did something that may seem non-obvious to another informed contributor. These conversations provide an opportunity for mentorship and nuanced discourse beyond the code review. They will help level-up your less experienced contributors and make them more confident and opinionated about how they write code in the future.

Automate

Automate as much of the above as possible (not the 1 on 1 conversations). Add linting rules and build-time enforcements that encourage the "right" types of contributions. Add minimum percentage thresholds for unit test coverage so that contributors don't accidentally forget to test their code. Systematically enforce structure where ever you can and adopt and socialize curated design architecture patterns within your team. As engineers become more familiar with it, this common structure pays dividends over time. Time writing code now gets spent on building features rather than trying to figure out how to contribute. Time in code review gets spent on refining implementations rather than quibbling over conventions. Make a contributor aware as soon as possible when they're diverging from a pattern or standard you want to maintain. Lastly, be empathetic and assume all contributions are made in good faith. You'll find that standards for "good code" differ between individuals and companies, so when you find someone who does things differently than you, understand that you may disagree on standards, but do align on some standards and commit to abide by them.

quality

teams

What's in this posts