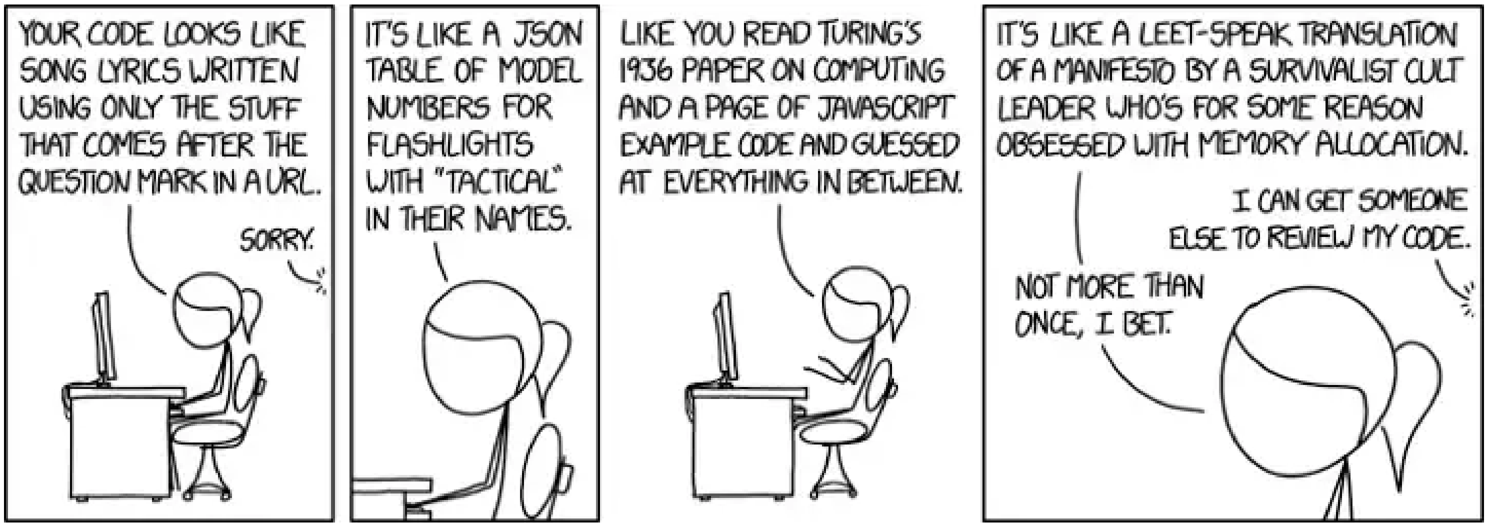
Sourcery Follow

Jan 23 · 5 min read

Code reviews need to shift-left


YOUR CODE LOOKS LIKE SONG LYRICS WRITTEN USING ONLY THE STUFF THAT COMES AFTER THE QUESTION MARK IN A URL.
SORRY.
IT'S LIKE A JSON TABLE OF MODEL NUMBERS FOR FLASHLIGHTS WITH "TACTICAL" IN THEIR NAMES.
LIKE YOU READ TURING'S 1936 PAPER ON COMPUTING AND A PAGE OF JAVASCRIPT EXAMPLE CODE AND GUESSED AT EVERYTHING IN BETWEEN.
IT'S LIKE A LEET-SPEAK TRANSLATION OF A MANIFESTO BY A SURVIVALIST CULT LEADER WHO'S FOR SOME REASON OBSESSED WITH MEMORY ALLOCATION.
I CAN GET SOMEONE ELSE TO REVIEW MY CODE.
NOT MORE THAN ONCE, I BET.

XKCD on code reviews — https://xkcd.com/1513/

We often talk about the idea of "shifting-left" for testing code and security analysis to speed up development and reduce risks further on the software engineering lifecycle. Moving testing and security analysis and improvements as early on in the development process as possible help to catch issues early and cut down on rework — making everyone's lives easier. But, testing and security analysis aren't the only places that we should taking this approach — we also can shift more and more of the overall code review process further left.

Now, I'm not saying that we should abolish the idea of reviewing code in any way. There's a reason code reviews have become an integral part of most development processes. They can be very helpful to:

Determine if new code is doing the right thing: At the end of the day we need to figure out if our new code is actually doing the right thing for our project or our business. If it's not then the entire exercise around improving development efficiency is pretty irrelevant.

Decide if the right structure was chosen: Even if the code is doing the *right* thing, its structure may be ineffective or inefficient.

Check for best practices and common pitfalls: New functionality could be designed the right way at a high level, but could still not fit into the broader best practices and styles of the rest of the project. If the new code doesn't follow these standards then technical debt can quickly build up — making it more difficult to understand the codebase and slowing down future development.

How does the code fit into the rest of the project: Even following all of the style and best practices of the team the new code could still have architectural or design flaws in terms of the overall project structure.

Share knowledge across the team: A codebase is continually evolving — and the full team needs to be kept up to speed about how it is changing. Things like documentation can help with knowledge sharing, but reviewing new changes also disseminates this information.

But, there's no reason that these things need to be done in a traditional async code review process. Each of these benefits of async or traditional code reviews can be tackled by a combination of pair or mob programming and automated tooling.

Checking for correctness, knowledge sharing, and design decisions are all arguably easier to check for when multiple developers are working together rather than in full isolation. By working together there's more explicit discussion around best approaches, why design decisions were made, and ensuring that the best approach was taken.

Best practices, styles, and standardization can all be tackled by a variety of automation tooling throughout the development process. Code styling tools, linting tools, and refactoring tools can combine together to review changes to the code automatically — giving quick insight on where there are issues and inconsistencies in the code. And more and more tools are providing ways to fix these issues in your code along with flagging them.

The Issues with Async Reviews

Asynchronous code reviews have two major disadvantages that can prove quite costly to teams.

1. Context Gaps. The reviewer doesn't have the full context of how and why decisions were made for the code they were reviewing. As they review they may try to dive in and understand these decisions — but they won't be devoting as much time during the review

as the developer who first wrote the code. As a result they can miss out on some of the context for why decisions were made, limiting knowledge sharing and, more critically, running the risk that issues can fall through the cracks.
You could always argue that there's an advantage to having a limited context reviewer because that's the situation that most future developers will be in (the vast majority of time spent interacting with code is focused on reading code you didn't right), but the cons here far outweigh the pros.

2. Time Lags. It can take hours (or days) for code reviews to be completed. If multiple reviews are needed this can quickly cause the delivery time for a project to quickly expand. On top of this, a reviewer needs to switch context to understand what they're reviewing every time they approach a new review.

Async reviews do have their advantages as well, and it's worth acknowledging them:

1. Reduced Chance of Groupthink. Having a reviewer external to the initial development forces someone with a new perspective to consider the code and may uncover poor decisions that the initial developer or group made.

2. Outcome, Not Process Focused. With the reviewer primarily focused on the outcome of the code (eg. what the new feature does, whether the bug was fixed, etc) they are able to be more objective and less caught up in the details around the process to get there (what tools were used, intermediate discussions, etc)

Ultimately the large time costs and context issues around async reviews don't outweigh the benefits — and teams can actively work to make sure they are continually focused on the optimal outcome throughout their development process to counteract issues around groupthink and an over-focus on the process.

Shifting away from an asynchronous code review process can seem like a risky or arduous undertaking. But, moving more of your code reviews earlier into the process shortens the time to feedback and can improve the actual outcomes from the code reviews themselves. And, it doesn't need to be an all or nothing approach — you can start to use more pair programming alongside automated tooling while retaining a final manual, async, code review check to make sure nothing is falling through the cracks — and then move to a more continual review and pairing driven development process.

See how Sourcery is working to help development teams automatically review and improve their code throughout the software development lifecycle. From language standards to project specific best practices *you can continually improve the quality of the code your team is working with.*

Code Review Code Quality Software Development