

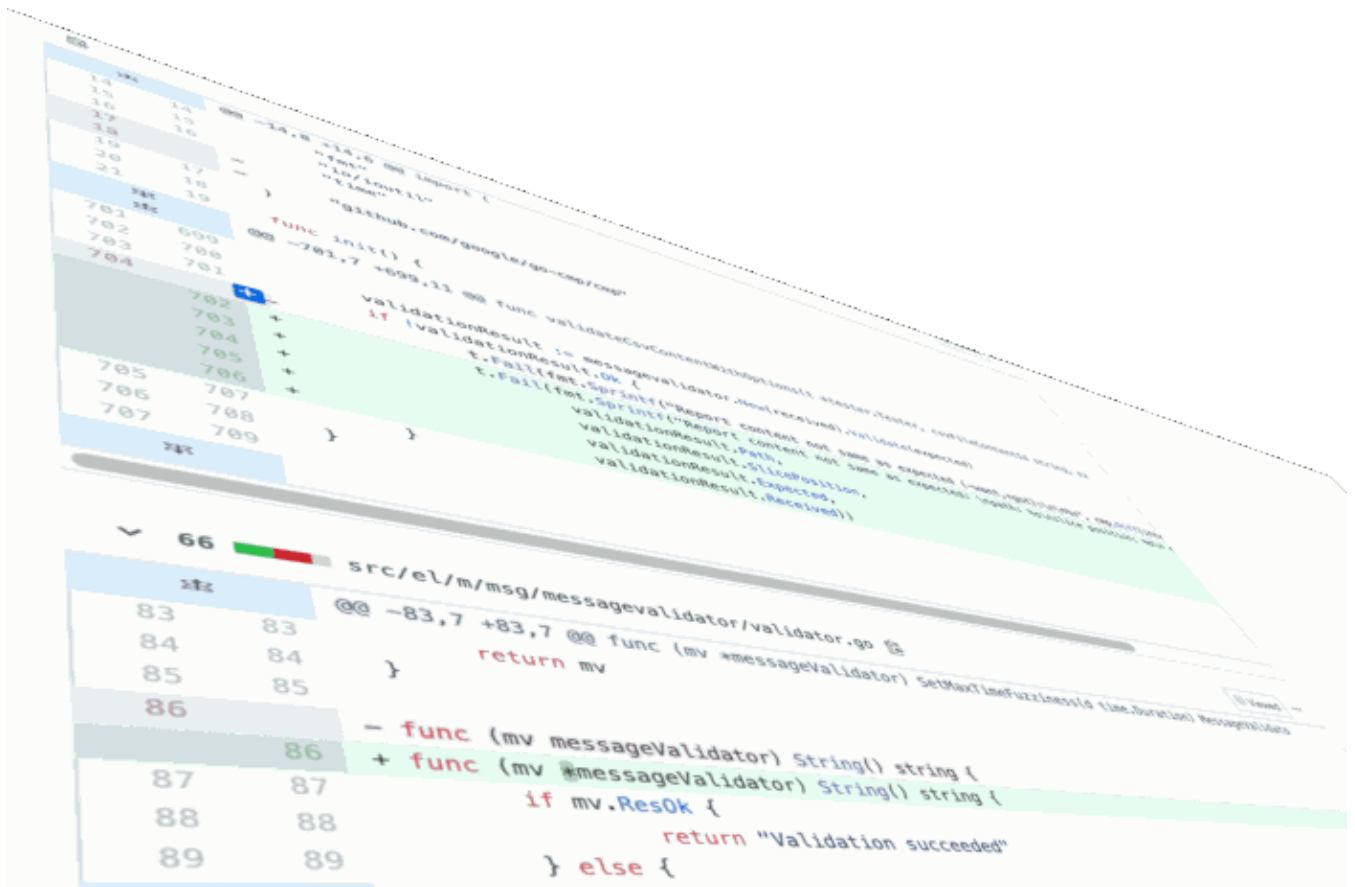
# How one code review rule turned my team into a dream team

Time reviewing code is time well spent.



Elena Flat

Apr 17 · 8 min read



Consider the following scenario:

*Bob the product owner slacks the team of three developers - Alice, Caithe, Duke - about urgent modifications needed for the report they just delivered. There is a client meeting the next day at 10am, and changes have to be live so that Bob can demo. It is 6pm. Alice is commuting home, Caithe is on the way to a cottage (she took some time off). Duke however happens to be at his laptop.*

Here's what happens next:

- Duke sees Bob's message and lets Bob know that, no problem, he can make the changes. However he won't be available starting 9pm and all morning tomorrow as he has family commitments. Thus he may not be able to push the changes live or support in the morning.
- Caithe, being the passenger in the back seat, sees this on Slack, and says "no problem, I'll be there in the morning if anything needs finishing".
- Duke starts writing some code and opens a pull request (PR).
- At 8pm, Alice gets home, looks at Slack, and lets the team know she can help out now. She takes a look at Duke's PR, realizes there are no tests as this is a rush job, and decides to add tests. Since requirements are on Slack and they can collaborate on each other's forks, this is not a problem.
- Duke drops off at 9pm.
- At 10pm, Alice finishes the tests, by which time Caithe is online and able to review. Caithe catches a bug (Alice reversed some requirements in her head, as she's pretty sleepy), and after Alice tweaks the code, they push it live.
- Come morning, Caithe makes a few more changes due to some *very* last minute additional requirements from Bob, and pushes them live. Duke approves the second PR from his phone (he was worried and was checking in). Client meeting is a smashing success.

Reverse the genders, and this is basically what happened on my team one late Thursday night. I felt very proud. Unusually though, it wasn't that silly pride I used to feel when I was 24 and pulling all-nighters for a startup: *Look at me, my job is very important and if I don't work 24h straight the world will end.*

This time my pride was fuelled by different things:

- All three developers on my team were equally familiar with the code and equally equipped to make modifications and review the work.
- The entire team rose to the occasion, not leaving any one person to work late night.
- Because there were 3 of us, we could do a high quality job even when rushed

As result:

- There was full test coverage, even though it was an urgent piece of work. Thus no dev-debt.
- All the bugs were caught by reviews.
- Team camaraderie and confidence in each other was at an all-time high!

. . .

I am coming up to a year now on the same team, and by this time you get a bit too familiar with the code, so you start having thoughts: *should I switch teams?* Now, there are different reasons to consider: are you being challenged? What's on the roadmap? What are your career plans? I will not go into any of that here. What I want to talk about is why I love working on this team in particular, so much so that I don't want to switch.

Having thought about it, I realized I love working on my team not because of *who* is on it or *what* the team works on. I love *how* we work:

- The entire team knows most of the codebase we own, so we do all discussions / design & architecture together. This leads to a very good variety of ideas and robust technical designs (we happen to have a pretty diverse team, luckily for us).
- We can swap out in emergencies when some of us are unavailable.
- We refactor and experiment often — and feel safe doing it.

How did we get there? **The Pull Request review rules and guidelines we established for our team.**

When the team was formed, we were just 3 developers, a product owner and half a manager (the other half of our manager was much too busy managing a bigger team). We decided that since there are so few of us, we might as well function as one. So we came up with a PR review rule: **Whatever one of us writes, the other 2 have to be cool with.** Then, as we kept reviewing each other's PRs consistently, we expanded the ruleset to a few guidelines.

I am writing down these rules and guidelines here, as I realized maybe some people will find them useful for their teams. Or maybe these will give you some food for thought and you come up with completely different ideas.

## Rule 1

---

***Each PR review must have at least 2 same-team developer approvals. Manager approval does not count.***

---

The first thing to note here is since we started off as a team of 3, this was ideal. All 3 developers were in-the-know 100% of the time. With larger teams this could be different. What you are aiming for is a Dumbledore-Horcruxes situation, where if you die, at least 2 or 3 people know about Horcruxes. Our team is 5 developers now, with 2 old timers, 1 new comer, and 2 in-the-middle. We are still trying to stick to a model where everyone reviews everyone's PRs, but it might make sense to switch to 1 old timer + 1 other developer minimum.

“Manager approval does not count” — maybe this made you think “WHAT? I am a manager, and an amazing coder, and my review totally counts, how dare you.”

Manager review does *help*. We have amazing managers who happen to be great coders, and when they do have time to review, they often have great tips or ideas to help us out with.

But you have to consider how much time your manager spends coding. Is it more than 50%? Then they are also a developer, so okay, let's count them. At our company, however, dev managers do very occasional coding. At the end of the day, it is the developers who have to live with the code that's being written. Therefore developers get the final say. I really dislike seeing a manager-developer infinite approval cycle where a manager keeps approving one developer's PRs even though there are 3 other developers on the team who haven't had a chance to look at it yet.

## Rule 2

---

***Each PR must have a good description. From reading the description, the reviewer should be able to understand what the code is meant to do. This has to be true even if there is a Jira ticket or a requirements page.***

---

PRs without a description — on my team this will never go through. Best case, they didn't write it because the Jira ticket has a good description. Worst case: Jira ticket has

nothing.

The reviewer of a PR without a description has 2 jobs:

1. Understanding what the code does from reading the code changes in the PR
2. Trying to decide if the code does *what it's supposed to do* based on... the reviewer's general views on the universe and the current geopolitical climate?

If there's no statement that clearly explains what the code is meant to do, reviewing it for correctness is not possible. You simply don't know what is correct. You operate on assumptions of what *you think* is correct.

## Rule 3

***PR must have sufficient unit test and integration test coverage.***

Ideally the reviewer has an easy way to see the list of test cases covered.

If test coverage already exists, or for whatever reason test coverage is not complete (work split off into another ticket, dependencies), PR description should mention this.

**Giving good PR reviews is hard and time consuming.** That's because once you understand what the code is supposed to do, you have to stop reviewing and jot down what test coverage *you'd* want for this. You make *your own* test cases list and decide what kind they are: unit test/integration/end-to-end.

You also have to think if there are possible production implications, and if any testing on live data is needed pre-deploy.

Now that you know all this, you go back to reviewing the PR and check if they covered everything you thought of. Ideally it's a mix — you thought of something they didn't, and they thought of something you didn't, and together you do a good job. This is where being able to easily see a list of test cases covered in PR comes in handy (vs having to read 100s or 1000s of lines of tests and have to figure out what cases are actually being tested).

Note that if Rule 2 has been followed, you can tell what the code is supposed to do just from reading the description. So you can do the step of reviewing test coverage before you've reviewed any actual code!

I personally like to keep all this — mention of tests, any live data testing, any live client implications/prod config changes needed — on the PR description: That way it's in one place. Could be a linked Jira ticket. As long as it's somewhere and has been considered.

## Rule 4

*If the PR is a bug fix, it must contain a test such that, should the bug fix be reverted, this test would fail.*

I thought this was obvious, but as with most obvious things — it's not.

Say there's a typo bug where you went `!=` instead of `==`. It was 10pm and you shouldn't have been coding anyway, but you felt like going the extra mile that night. So you open a bug fix PR and change `!=` to `==`. Some kind soul, also working, approves it, and off it goes to the master branch.

Three days later your buddy Duke is finally ready with his PR that got out of control: he thought it was a small feature and didn't break it into smaller tickets, but that didn't turn out to be the case and now he has a 2000 line PR he is begging his team to review.

Duke merges from upstream, the code conflicts where you swapped `==` in for `!=`. Duke is tired, he doesn't notice what's up, `!=` is back.

Since there's no test to catch it, the bug's back.

This is a contrived example, but if you pay attention over longer periods — this happens all the time. Lay out the bugs over 3 years, you'll see those not trapped by a test reappearing.

. . .


That's it! Three rules, and fourth one that's more of a common sense item.


These rules, I realized, led to our team producing some pretty good quality code. I am very happy about that, and by no means take the credit.

Again, these rules might feel wrong to you, or maybe they happen to be helpful. If they sparked any thoughts or ideas, I'm happy with that!

[Code Review](#)[Code Quality](#)[Teamwork](#)[Coding](#)[Programming](#)[About](#) [Help](#) [Legal](#)

### Get the Medium app

 A button that says 'Download on the App Store', and if clicked it will lead you to the iOS App store

 A button that says 'Get it on, Google Play', and if clicked it will lead you to the Google Play store