Kevin London's blog About

Code Review Best Practices

May 5, 2015

At my current company, we do a fair amount of code reviews. I had never done one before I started here so it was a new experience for me. I think it's a good idea to crystalize some of the things I look for when I'm doing code reviews and talk about the best way I've found to approach them.

Briefly, a code review is a discussion between two or more developers about changes to the code to address an issue. Many articles talk about the benefits of code reviews, including knowledge sharing, code quality, and developer growth. I've found fewer that talk about what to look for in a review and how to discuss code under review.

What I look for during a review

Architecture / Design

- **Single Responsibility Principle:** The idea that a class should have one-and-only-one responsibility. Harder than one might expect. I usually apply this to methods too. If we have to use "and" to finish describing what a method is capable of doing, it might be at the wrong level of abstraction.
- **Open/Closed Principle:** If the language is object-oriented, are the objects open for extension but closed for modification? What happens if we need to add another one of `x` ?
- **Code duplication:** I go by the "three strikes" rule. If code is copied once, it's usually okay although I don't like it. If it's copied again, it should be refactored so that the common functionality is split out.
- **Squint-test offenses:** If I squint my eyes, does the shape of this code look identical to other shapes? Are there patterns that might indicate other problems in the code's structure?
- **Code left in a better state than found:** If I'm changing an area of the code that's messy, it's tempting to add in a few lines and leave. I recommend going one step further and leaving the code nicer than it was found.
- **Potential bugs:** Are there off-by-one errors? Will the loops terminate in the way we expect? Will they terminate at all?
- **Error handling:** Are errors handled gracefully and explicitly where necessary? Have custom errors been added? If so, are they useful?
- **Efficiency:** If there's an algorithm in the code, is it using an efficient implementation? For example, iterating over a list of keys in a dictionary is an inefficient way to locate a desired value.

Style

- **Method names:** Naming things is one of the hard problems in computer science. If a method is named `get_message_queue_name` and it is actually doing something completely different like sanitizing HTML from the input, then that's an inaccurate method name. And probably a misleading function.
- **Variable names:** `foo` or `bar` are probably not useful names for data structures. `e` is similarly not useful when compared to `exception` . Be as verbose as you need (depending on the language). Expressive variable names make it easier to understand code when we have to revisit it later.
- **Function length:** My rule of thumb is that a function should be less than 20 or so lines. If I see a method above 50, I feel it's best that it be cut into smaller pieces.
- **Class length:** I think classes should be under about 300 lines total and ideally less than 100. It's likely that large classes can be split into separate objects, which makes it easier to understand what the class is supposed to do.
- **File length:** For Python files, I think around 1000 lines of code is about the most we should have in one file. Anything above that is a good sign that it should be split into smaller, more focused files. As the size of a file goes up, discoverability goes down.
- **Docstrings:** For complex methods or those with longer lists of arguments, is there a docstring explaining what each of the arguments does, if it's not obvious?
- **Commented code:** Good idea to remove any commented out lines.
- **Number of method arguments:** For the methods and functions, do they have 3 or fewer arguments? Greater than 3 is probably a sign that it could be grouped in a different way.
- **Readability:** Is the code easy to understand? Do I have to pause frequently during the review to decipher it?

Testing

- **Test coverage:** I like to see tests for new features. Are the tests thoughtful? Do they cover the failure conditions? Are they easy to read? How fragile are they? How big are the tests? Are they slow?

- **Testing at the right level:** When I review tests I'm also making sure that we're testing them at the right level. In other words, are we as low a level as we need to be to check the expected functionality? Gary Bernhardt recommends a ratio of 95% unit tests, 5% integration tests. I find that particularly with Django projects, it's easy to test at a high level by accident and create a slow test suite so it's important to be vigilant.
- **Number of Mocks:** Mocking is great. Mocking everything is not great. I use a rule of thumb where if there's more than 3 mocks in a test, it should be revisited. Either the test is testing too broadly or the function is too large. Maybe it doesn't need to be tested at a unit test level and would suffice as an integration test. Either way, it's something to discuss.
- **Meets requirements:** Usually as part of the end of a review, I'll take a look at the requirements of the story, task, or bug which the work was filed against. If it doesn't meet one of the criteria, it's better to bounce it back before it goes to QA.

Review your own code first

Before submitting my code, I will often do a `git add` for the affected files / directories and then run a `git diff --staged` to examine the changes I have not yet committed. Usually I'm looking for things like:

- Did I leave a comment or TODO in?
- Does that variable name make sense?
- …and anything else that I've brought up above.

I want to make sure that I would pass my own code review first before I subject other people to it. It also stings less to get notes from yourself than from others :p

How to handle code reviews

I find that the human parts of the code review offer as many challenges as reviewing the code. I'm still learning how to handle this part too. Here are some approaches that have worked for me when discussing code:

- **Ask questions:** How does this method work? If this requirement changes, what else would have to change? How could we make this more maintainable?
- **Compliment / reinforce good practices:** One of the most important parts of the code review is to reward developers for growth and effort. Few things feel better than getting praise from a peer. I try to offer as many positive comments as possible.
- **Discuss in person for more detailed points:** On occasion, a recommended architectural change might be large enough that it's easier to discuss it in person rather than in the comments. Similarly, if I'm discussing a point and it goes back and forth, I'll often pick it up in person and finish out the discussion.
- **Explain reasoning:** I find it's best both to ask if there's a better alternative and justify why I think it's worth fixing. Sometimes it can feel like the changes suggested can seem nit-picky without context or explanation.
- **Make it about the code:** It's easy to take notes from code reviews personally, especially if we take pride in our work. It's best, I find, to make discussions about the code than about the developer. It lowers resistance and it's not about the developer anyway, it's about improving the quality of the code.
- **Suggest importance of fixes:** I tend to offer many suggestions, not all of which need to be acted upon. Clarifying if an item is important to fix before it can be considered done is useful both for the reviewer and the reviewee. It makes the results of a review clear and actionable.

On mindset

As developers, we are responsible for making both working and maintainable code. It can be easy to defer the second part because of pressure to deliver working code. Refactoring does not change functionality by design, so don't let suggested changes discourage you. Improving the maintainability of the code can be just as important as fixing the line of code that caused the bug.

In addition, please keep an open mind during code reviews. This is something I think everyone struggles with. I can get defensive in code reviews too, because it can feel personal when someone says code you wrote could be better.

If the reviewer makes a suggestion, and I don't have a clear answer as to why the suggestion should not be implemented, I'll usually make the change. If the reviewer is asking a question about a line of code, it may mean that it would confuse others in the future. In addition, making the changes can help reveal larger architectural issues or bugs.

(Thanks to Zach Schipono for recommending this section be added)

Addressing suggested changes

We typically leave comments on a per-line basis with some thinking behind them. Usually I will look at the review notes in Stash and, at the same time, have the code pulled up to make the suggested changes. I find that I forget what items I am supposed to address if I do not handle them right away.

Additional References

There's a number of books on the art of creating clean code. I've read through fewer of these than I might like (and I'm working to change that). Here's a few books on my list:

- Clean Code by Robert C. Martin
- Refactoring by Martin Fowler, Kent Beck, and others

Some useful, related talks I'm a big fan of talks so here's a few that I thought of while writing this:

- All the Small Things by Sandi Metz: Covers the topic well, particularly from a perspective of writing clean, reusable code.
- How to Design a Good API and Why it Matters: API, in this sense, meaning the way in which the application is constructed and how we interact with it. Many of the points in the video talk about similar themes to those discussed here.

Discussion on Hacker News

Subscribe for more posts like this:

email address

Subscribe

Kevin London's blog

Kevin London's blog
kevinlondon@gmail.com

kevinlondon
@kevin_london

Software Engineer at Amazon. M.S. in Computer Science from Georgia Tech. Views are my own and not representative of my employer.