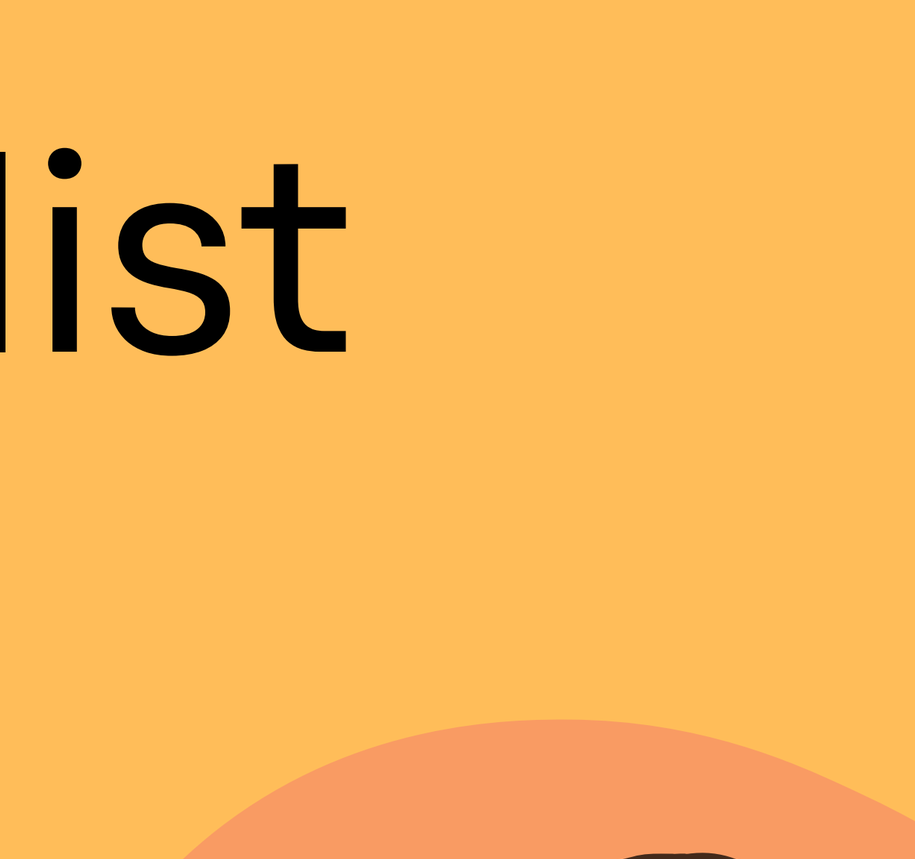
The Quick Start

Code Review Checklist

D·GOSLEN



A common question I get about code reviews is this:

> **What should I be looking for when reviewing code?**

And it's a great question.

My in-depth answer is you should be looking to gain understanding. We want to understand the code so well that we feel confident the code accomplishes its goal and that we are comfortable supporting it.

But still... where do you start?

This is my Quick Start Code Review Checklist for what to look for in a code review. This guide will help you quickly spot bugs, gaps, and areas for improvement that will move things along. No matter how little (or how much) code you have reviewed previously, this guide is for you.

No more wondering what to do when you get that notification to review some code.

Let's dive in!

This checklist is inspired by my book, Code Review Champion, so if you find it helpful - consider checking out the book!

1. Get Your Bearings

Whenever you start looking at new code, you have to get your bearings first.

When we first see a set of changes - via a git diff, pull request, or just on someone else's screen - we first need to understand how these changes fit into our mental model of the code. We must know what classes, modules, methods, etc., are changing. We also need to understand how these changes interact with the other modules and parts of the system that haven't changed.

An excellent way to get your bearings is to read all the changes before diving into specific details.

Yes - all of them.

Reading all of the changes upfront will help you build a good enough mental model of what is happening. You'll be able to see both the parts and the whole well. If you still need more understanding, you should look beyond the immediate code changes and review the entire codebase associated with the change.

It might sound tedious, but the results are worth it.

2. Gain Some Context

Once we have our bearings, we need to know what the changes are trying to accomplish to be sure our feedback is accurate and helpful.

Here are a few core questions that we need answers to at the start of any code review:

- What is the goal of these changes? What are the business requirements (or technical needs) that prompted them?
- Do these changes accomplish that goal?
- Are these changes moving us closer to our technical vision?

These questions will help frame your understanding of the changes and what they are trying to accomplish. You can even templatize these questions and include them in a pull request template so they get asked every time an author opens a pull request!

Don't be bashful to ask more questions and gain clarity on the code.

The sooner you can get aligned, the better the review will go.

3. Common Gotchas

There are a handful of common 'gotchas' in programming that we have to look out for.

This guide doesn't go into depth on each one, but these are definitely things to pay attention to.

1. Code Comments

Make sure any comments explain *why* rather than just repeating what.

If code comments are not aligned with what the code is doing, make sure they get aligned before the code is merged.

Too many comments is usually a sign the code is too fragile or complex. Try to understand if you can simplify it.

2. Plurality

Make sure the plurality of a variable name matches its type. An array or a collection should be plural; something that isn't shouldn't be.

It seems simple (and it is!), but it gets missed more than you might imagine.

3. Regular Expressions (or other complex DSLs)

Anytime you use a regular expression or some cryptic DSL syntax, be sure to explain what it is doing.

This goes against the idea earlier that comments should explain why and not what. But every rule usually has an exception. This is that exception.

Explain what the regex is doing. Detail each component and how it works. Give examples of what it matches or doesn't match.

Your fellow teammates (and future self!) will be glad the expression is well-documented.

4. Breaking Language Constructs

Every programming language has its set of pros and cons. And its own way of expressing or accomplishing things.

Java code doesn't look or feel like Python, and neither does Typescript feel like golang.

Pay attention to what idiomatic code in the language you are using feels like. Be sure to speak up whenever you see code that is drifting from those idioms and conventions.

It might seem pedantic, but I promise that code that "feels" like the language it's written in will help developer productivity and understanding.

5. Library Misuse or Misunderstandings

The same issues around properly using the coding language of choice extend to libraries and frameworks.

Few applications exist today without some other framework or library being pulled from the open-source community. And while that is good, it has its own pitfalls.

Libraries and frameworks can be large, confusing, and change often. Our code reviews must verify that the code correctly utilizes such libraries and frameworks.

In order to know if a library is being used properly, you need to have a good understanding of it works. Read the documentation all the way through if you can. If that isn't feasible, focus on the documents of APIs the code under review is using.

If you see code incorrectly utilizing these libraries, speak up! State the misuse and provide the documentation as evidence as to why. Work with the author to correct it accordingly.

Remember that libraries and frameworks are pretty expansive and deep. It could be that **you** misunderstand something. This is a great place to ask questions instead of statements.

Not only will questions act as a hedge against being wrong, but they will lead to better conversations where learning can happen.

6. Concurrency

Concurrency is hard. Very hard.

Few engineers can get concurrent algorithms and code to work the way we want in their first few attempts. There are edge cases and nuances in concurrent code that can get missed by a programmer of any level.

When you see concurrent code, do your best to think like the CPU. The CPU can only do what the code tells it to do - no more and no less. Think about the specific instructions and changes rather than the general purpose of the code.

Try to develop edge cases ("what happens if...") and walk through the code slowly utilizing techniques like rubber ducking or pair programming. If you have one available, grab a huge whiteboard, step through the code across a few threads working simultaneously, and see what happens.

Review each test with scrutiny and skepticism. Run the tests repeatedly to see if they fail at any point.

Concurrent code is one of the few places where you really **want** to find a bug vs. just trying to understand what is happening.

Take the time you need to review it properly.

7. Tests

Software should have tests. The exact amount and type of tests often get debated back and forth. From Thought Works' Testing Pyramid to Kent C. Dodds' Testing Trophy, there are lots of ideas about how to best do software testing.

When reviewing code, you need to make sure the code has proper tests. Code that isn't tested properly will only burden your team as the code becomes more and more complex.

The important keyword here is proper. Lots of bad tests don't do much good.

Review which use case each test is attempting to verify. Is it in line with the business requirements? Does it have proper assertions? Are there assumptions being made? By paying special attention to the tests, you get the "free" benefit of double-checking the expectations and requirements of the system code. A win-win.

Lastly, don't insist on too many tests. Every line of code written is a line of code to maintain.

But there should be tests and there should be enough of them that give you the confidence they will catch bugs in the software; now or in the future

Hooray! You did it!

Thanks for hanging in there and reading it all the way through.

Code reviewing is tough, but I believe this checklist can help you know where to focus your time and energy. Knowing where to start and what to look for is half the game.

I hope it helps you in your code review journey!

If you enjoyed this checklist, check out my book Code Review Champion!

Let's connect!

D·GOSLEN