Code Review Best Practices

Palantir
Mar 4, 2018 · 12 min read

The Internet provides a wealth of material on code reviews: on the effect of code reviews on company culture, on formal security reviews, shorter guides, longer checklists, humanized reviews, reasons for doing code reviews in the first place, best practices, more best practices, statistics on code review effectiveness for catching bugs, and examples of code reviews gone wrong. Oh, and of course there are books, too. Long story short, this blog post presents Palantir's take on code reviews. Organizations with deep cultural reluctance to peer reviews may want to consult Karl E. Wiegers' excellent essay on Humanizing Peer Reviews before trying to follow this guide.

This post is copied from the best practices guides of our Java Code Quality tool chain, Baseline, and covers the following topics:

- Why, what, and when to do code reviews
- Preparing code for review
- Performing code reviews
- Code review examples

Motivation

We perform code reviews (CRs) in order to improve code quality and benefit from positive effects on team and company culture. For example:

- **Committers are motivated** by the notion of a set of reviewers who will look over the change request: the committer tends to clean up loose ends, consolidate TODOs, and generally improve the commit. Recognition of coding expertise through peers is a source of pride for many programmers.
- **Sharing knowledge** helps development teams in several ways:
  - A CR explicitly communicates added/altered/removed functionality to team members who can subsequently build on the work done.
  - The committer may use a technique or algorithm that reviewers can learn from. More generally, code reviews help raise the quality bar across the organization.
  - Reviewers may possess knowledge about programming techniques or the code base that can help improve or consolidate the change; for example, someone else may be concurrently working on a similar feature or fix.
  - Positive interaction and communication strengthens social bonds between team members.
- **Consistency** in a code base makes code easier to read and understand, helps prevent bugs, and facilitates collaboration between regular and migratory developer species.
- **Legibility** of code fragments is hard to judge for the author whose brain child it is, and easy to judge for a reviewer who does not have the full context. Legible code is more reusable, bug-free, and future-proof.
- **Accidental errors** (e.g., typos) as well as **structural errors** (e.g., dead code, logic or algorithm bugs, performance or architecture concerns) are often much easier to spot for critical reviewers with an outside perspective. Studies have found that even

short and informal code reviews have significant impact on code quality and bug frequency.

- **Compliance** and regulatory environments often demand reviews. CRs are a great way to avoid common security traps. If your feature or environment has significant security requirements it will benefit from (and probably require) review by your local security curmudgeons (OWASP's guide is a good example of the process).

What to review

There is no eternally true answer to this question and each development team should agree on its own approach. Some teams prefer to review every change merged into the main branch, while others will have a "triviality" threshold under which a review is not required. The trade-off is between effective use of engineers' (both authors' and reviewers') time and maintaining code quality. In certain regulatory environments, code review may be required even for trivial changes.

Code reviews are classless: being the most senior person on the team does not imply that your code does not need review. Even if, in the rare case, code is flawless, the review provides an opportunity for mentorship and collaboration, and, minimally, diversifies the understanding of code in the code base.

When to review

Code reviews should happen after automated checks (tests, style, other CI) have completed successfully, but before the code merges to the repository's mainline branch. We generally don't perform formal code review of aggregate changes since the last release.

For complex changes that should merge into the mainline branch as a single unit but are too large to fit into one reasonable CR, consider a stacked CR model: Create a primary branch `feature/big-feature` and a number of secondary branches (e.g., `feature/big-feature-api`, `feature/big-feature-testing`, etc.) that each encapsulate a subset of the functionality and that get individually code-reviewed against the `feature/big-feature` branch. Once all secondary branches are merged into `feature/big-feature`, create a CR for merging the latter into the main branch.

Preparing code for review

It is the author's responsibility to submit CRs that are easy to review in order not to waste reviewers' time and motivation:

- **Scope and size**. Changes should have a narrow, well-defined, self-contained scope that they cover exhaustively. For example, a change may implement a new feature or fix a bug. Shorter changes are preferred over longer ones. If a CR makes substantive changes to more than ~5 files, or took longer than 1–2 days to write, or would take more than 20 minutes to review, consider splitting it into multiple self-contained CRs. For example, a developer can submit one change that defines the API for a new feature in terms of interfaces and documentation, and a second change that adds implementations for those interfaces.

- Only submit **complete, self-reviewed** (by diff), and **self-tested** CRs. In order to save reviewers' time, test the submitted changes (i.e., run the test suite) and make sure they pass all builds as well as all tests and code quality checks, both locally and on the CI servers, *before assigning reviewers*.

- **Refactoring changes** should not alter behavior; conversely, a behavior-changing changes should avoid refactoring and code formatting changes. There are multiple good reasons for this:
  - Refactoring changes often touch many lines and files and will consequently be reviewed with less attention. Unintended behavior changes can leak into the code base without anyone noticing.
  - Large refactoring changes break cherry-picking, rebasing, and other source control magic. It is very onerous to undo a behavior change that was introduced as part of a repository-wide refactoring commit.
  - Expensive human review time should be spent on the program logic rather than style, syntax, or formatting debates. We prefer settling those with automated tooling like Checkstyle, TSLint, Baseline, Prettier, etc.

Commit messages

The following is an example of a good commit message following a widely quoted standard:

```
Capitalized, short (80 chars or less) summary

More detailed explanatory text, if necessary. Wrap it to about 120
characters or so. In some contexts, the first
line is treated as the subject of an email and the rest of the text
```

```
as the body. The blank line separating the
summary from the body is critical (unless you omit the body
entirely); tools like rebase can get confused if you run
the two together.

Write your commit message in the imperative: "Fix bug" and not
"Fixed bug" or "Fixes bug." This convention matches
up with commit messages generated by commands like git merge and git
revert.

Further paragraphs come after blank lines.

- Bullet points are okay, too
```

Try to describe both what the commit changes and how it does it:

```
> BAD. Don't do this.
Make compile again

> Good.
Add jcsv dependency to fix IntelliJ compilation
```

Finding reviewers

It is customary for the committer to propose one or two reviewers who are familiar with the code base. Often, one of the reviewers is the project lead or a senior engineer. Project owners should consider subscribing to their projects in order to get notified of new CRs. Code reviews among more than three parties are often unproductive or even counter-productive since different reviewers may propose contradictory changes. This may indicate fundamental disagreement on the correct implementation and should be resolved outside a code review in a higher-bandwidth forum, for example in person or in a video conference with all involved parties.

Performing code reviews

A code review is a synchronization point among different team members and thus has the potential to block progress. Consequently, code reviews need to be prompt (on the order of hours, not days), and team members and leads need to be aware of the time commitment and prioritize review time accordingly. If you don’t think you can complete a review in time, please let the committer know right away so they can find someone else.

A review should be thorough enough that the reviewer could explain the change at a reasonable level of detail to another developer. This ensures that the details of the code base are known to more than a single person.

As a reviewer, it is your responsibility to enforce coding standards and keep the quality bar up. Reviewing code is more of an art than a science. The only way to learn it is to do it; an experienced reviewer should consider putting other less experienced reviewers on their changes and have them do a review first. Assuming the author has followed the guidelines above (especially with respect to self-review and ensuring the code runs), here's an list of things a reviewer should pay attention to in a code review:

Purpose

- **Does this code accomplish the author's purpose?** Every change should have a specific reason (new feature, refactor, bugfix, etc). Does the submitted code actually accomplish this purpose?

- **Ask questions.** Functions and classes should exist for a reason. When the reason is not clear to the reviewer, this may be an indication that the code needs to be rewritten or supported with comments or tests.

Implementation

- **Think about how you would have solved the problem.** If it's different, why is that? Does your code handle more (edge) cases? Is it shorter/easier/cleaner/faster/safer yet functionally equivalent? Is there some underlying pattern you spotted that isn't captured by the current code?

- **Do you see potential for useful abstractions?** Partially duplicated code often indicates that a more abstract or general piece of functionality can be extracted and then reused in different contexts.

- **Think like an adversary, but be nice about it.** Try to "catch" authors taking shortcuts or missing cases by coming up with problematic configurations/input data that breaks their code.

- **Think about libraries or existing product code.** When someone re-implements existing functionality, more often than not it's simply because they don't know it already exists. Sometimes, code or functionality is duplicated on purpose, e.g., in order to avoid dependencies. In such cases, a code comment can clarify the intent. Is the introduced functionality already provided by an existing library?

- **Does the change follow standard patterns?** Established code bases often exhibit patterns around naming conventions, program logic decomposition, data type definitions, etc. It is usually desirable that changes are implemented in accordance with existing patterns.

- **Does the change add compile-time or run-time dependencies (especially between sub-projects)?** We want to keep our products loosely coupled, with as few dependencies as possible. Changes to dependencies and the build system should be scrutinized heavily.

Legibility and style

- **Think about your reading experience.** Did you grasp the concepts in a reasonable amount of time? Was the flow sane and were variable and methods names easy to follow? Were you able to keep track through multiple files or functions? Were you put off by inconsistent naming?

- **Does the code adhere to coding guidelines and code style?** Is the code consistent with the project in terms of style, API conventions, etc.? As mentioned above, we prefer to settle style debates with automated tooling.

- **Does this code have TODOs?** TODOs just pile up in code, and become stale over time. Have the author submit a ticket on GitHub Issues or JIRA and attach the issue number to the TODO. The proposed code change should not contain commented-out code.

Maintainability

- **Read the tests.** If there are no tests and there should be, ask the author to write some. Truly untestable features are rare, while untested implementations of features are unfortunately common. Check the tests themselves: are they covering interesting cases? Are they readable? Does the CR lower overall test coverage? Think of ways this code could break. Style standards for tests are often different than core code, but still important.

- **Does this CR introduce the risk of breaking test code, staging stacks, or integrations tests?** These are often not checked as part of the pre-commit/merge checks, but having them go down is painful for everyone. Specific things to look for are: removal of test utilities or modes, changes in configuration, and changes in artifact layout/structure.

- **Does this change break backward compatibility?** If so, is it OK to merge the change at this point or should it be pushed into a later release? Breaks can include database or schema changes, public API changes, user workflow changes, etc.

- **Does this code need integration tests?** Sometimes, code can't be adequately tested with unit tests alone, especially if the code interacts with outside systems or configuration.

- **Leave feedback on code-level documentation, comments, and commit messages.** Redundant comments clutter the code, and terse commit messages mystify future contributors. This isn't always applicable, but quality comments and commit messages will pay for themselves down the line. (Think of a time you saw an excellent, or truly terrible, commit message or comment.)

- **Was the external documentation updated?** If your project maintains a README, CHANGELOG, or other documentation, was it updated to reflect the changes? Outdated documentation can be more confusing than none, and it will be more costly to fix it in the future than to update it now.

Don't forget to praise concise/readable/efficient/elegant code. Conversely, declining or disapproving a CR is not rude. If the change is redundant or irrelevant, decline it with an explanation. If you consider it unacceptable due to one or more fatal flaws, disapprove it, again with an explanation. Sometimes the right outcome of a CR is "let's do this a totally different way" or even "let's not do this at all."

Be respectful to the reviewees. While adversarial thinking is handy, it's not your feature and you can't make all the decisions. If you can't come to an agreement with your reviewee with the code as is, switch to real-time communication or seek a third opinion.

Security

Verify that API endpoints perform appropriate authorization and authentication consistent with the rest of the code base. Check for other common weaknesses, e.g., weak configuration, malicious user input, missing log events, etc. When in doubt, refer the CR to an application security expert.

Comments: concise, friendly, actionable

Reviews should be concise and written in neutral language. Critique the code, not the author. When something is unclear, ask for clarification rather than assuming ignorance. Avoid possessive pronouns, in particular in conjunction with evaluations: "*my* code worked before *your* change", "*your* method has a bug", etc. Avoid absolute judgements: "this can *never* work", "the result is *always* wrong".

Try to differentiate between suggestions (e.g., "Suggestion: extract method to improve legibility"), required changes (e.g., "Add @Override"), and points that need discussion or clarification (e.g., "Is this really the correct behavior? If so, please add a comment explaining the logic."). Consider providing links or pointers to in-depth explanations of a problem.

When you're done with a code review, indicate to what extent you expect the author to respond to your comments and whether you would like to re-review the CR after the changes have been implemented (e.g., "Feel free to merge after responding to the few minor suggestions" vs. "Please consider my suggestions and let me know when I can take another look.").

Responding to reviews

Part of the purpose of the code review is improve the author's change request; consequently, don't be offended by your reviewer's suggestions and take them seriously even if you don't agree. Respond to every comment, even if it's only a simple "ACK" or "done." Explain why you made certain decisions, why some function exists, etc. If you can't come to an agreement with the reviewer, switch to real-time communication or seek an outside opinion.

Fixes should be pushed to the same branch, but in a separate commit. Squashing commits during the review process makes it hard for the reviewer to follow up on changes.

Different teams have different merge policies: some teams allow only project owners to merge, while other teams allow the contributor to merge after a positive code review.

In-person code reviews

For the majority of code reviews, asynchronous diff-based tools such as Reviewable, Gerrit or, GitHub are a great choice. Complex changes, or reviews between parties with very different expertise or experience can be more efficient when performed in person, either in front of the same screen or projector, or remotely via VTC or screen share tools.

Examples

In the following examples, suggested review comments are indicated by `//R: ...` comments in the code blocks.

Inconsistent naming

```
class MyClass {
  private int countTotalPageVisits; //R: name variables consistently
  private int uniqueUsersCount;
}
```

Inconsistent method signatures

```
interface MyInterface {
  /** Returns {@link Optional#empty} if s cannot be extracted. */
  public Optional<String> extractString(String s);

  /** Returns null if {@code s} cannot be rewritten. */
  //R: should harmonize return values: use Optional<> here, too
  public String rewriteString(String s);
}
```

Library use

```
//R: remove and replace by Guava's MapJoiner
String joinAndConcatenate(Map<String, String> map, String
keyValueSeparator, String keySeparator);
```

Personal taste

```
int dayCount; //R: nit: I usually prefer numFoo over fooCount; up to
you, but we should keep it consistent in this project
```

Bugs

```
//R: This performs numIterations+1 iterations, is that intentional?
//   If it is, consider changing the numIterations semantics?
for (int i = 0; i <= numIterations; ++i) {
  ...
}
```

Architectural concerns

```
otherService.call(); //R: I think we should avoid the dependency on
OtherService. Can we discuss this in person?
```

. . .

Authors

Robert F (GitHub / Twitter)

Software Development Palantirtech Code Quality Code Review Software Engineering

Medium About Help Legal