Five Code Review Antipatterns

Everyone cares about best practices, but worst practices can sometimes be more illuminating.

by Trisha Gee

May 4, 2020

Code reviews are essential, but they are not always done correctly. This article points out, and rants about, particular antipatterns all developers have probably experienced while their code is reviewed or when they submit pull requests.

Antipattern: Nit-Picking

Imagine the following scenario. The code authors put hours, if not days, of effort into creating the solution they thought would work best. They considered multiple design options and took the path that seemed most relevant. They considered the architecture of the existing application and made changes in the appropriate places. Then they submitted their solution as a pull request or they started the code review process, and the expert feedback they received is

- “You should be using tabs, not spaces.”
- “I don’t like where the braces are in this section.”
- “You do not have a blank line at the end of your file.”
- “Your enums are in uppercase and they should be in sentence case.”

Although it’s important that new code be consistent with the style of existing code, these are hardly things that require a human reviewer. Human reviewers are expensive and can accomplish things computers cannot. Checking that style standards have been met is something a computer can easily do and distracts from the real purpose of a code review.

If developers see many of these types of comments during code review, it suggests the team either does not have a style guide or it has one but checking the style has not been automated. The solution is to use tools such as checkstyle to make sure style guidelines have been followed or to use sonarqube to identify common quality and security issues. Rather than relying on human reviewers to warn of such problems, the continuous integration environment can do that.

Sometimes, it might be difficult to automate such checks, for example, if no code guidelines exist or if the in-house code style has evolved over time and differs in various sections. There are approaches to applying...
automated checks in these situations. For example, a team can agree to
do a single commit that applies an agreed-upon code style and contains
no other changes. Or a team can agree that when a file is changed for a
bug or feature, the file will also be updated to the new style and the
automated tool can be configured to check only changed files.

If a team has a variety of code styles and it does not have a way to
automate checking the style, it’s also prone to falling into the next trap.

**Antipattern: Inconsistent Feedback**

For every developer invited to review a code, at least one more opinion is
invited—and probably more. Everyone can hold more than one opinion at
the same time. Sometimes a code review can descend into an argument
between reviewers about different approaches, such as whether streams
or a classic *for* loop would be best. How is a developer supposed to
make changes, close the review, and push the code to production if team
members have opposite views on the same code?

Even a single reviewer’s mind can easily change, either during one
review or over a series of reviews. On one review, a reviewer might be
pushing the author to make sure a data structure with $O(1)$ read
operations is used, whereas in the next review, the reviewer might ask
why there are several data structures for different use cases and suggest
the code should be simplified by doing a linear search through a single
structure.

This scenario happens when a team doesn’t have a clear idea of what its
“best practices” look like and when it hasn’t figured out what its priorities
are, for example:

- Should the code be moving towards a more modern style of Java?
  Or is it more important that the code is consistent and, therefore,
  continues to use “classic” constructs everywhere?

- Is it important to have $O(1)$ read operations on data structures in all
  parts of the system? Or are there sections where $O(n)$ is
  acceptable?

Almost all design questions can be answered with “it depends.” To have a
better idea of the answer, developers need to understand the priorities for
their applications and for their teams.

**Antipattern: Last-Minute Design Changes**

The most demoralizing feedback a developer can get during code review
is when a reviewer fundamentally disagrees with the design or
architecture of the solution and forces a complete rewrite of the code,
either gradually through a series of reviews (see the following section) or
by brutally rejecting the code and making the author start again.

Code review is not the right time to review the design. If the team is
following a classic “gateway” code review, the code should be working
and all the tests should be passing before the final step of having another
developer look at the code. At this point, hours, days, or possibly weeks
(although I really hope not weeks; code reviews should be small, but
that’s another subject) of effort have gone into the code under review.
Stating during code review that the underlying design is wrong is a waste
of everyone’s time.

Code reviews can be used as design reviews, but if this is the intention of
the code review, the review should be happening at the start of
implementation. Then, before developers get too far down the road, they
can sketch out their ideas with maybe some stub classes and methods
and some tests with meaningful names and steps, and perhaps submit some text or diagrams, in order to have team members give feedback on the approach to be taken.

If team members are finding genuinely showstopping design problems in a gateway review (that is, when the code is complete and working), the team should update its process to locate these problems much earlier. This might mean doing alternative types of reviews such as the one suggested in the previous paragraph, whiteboarding ideas, pair programming, or talking through the proposed solution with a tech lead. Finding design problems in a final code review is a waste of development time and enormously demoralizing for code authors.

**Antipattern: Ping-Pong Reviews**

In an ideal world, authors would submit their code for review, the reviewers would make a couple of comments with clear solutions, the authors would make the suggested changes and resubmit the code, the review would be complete, and the code would be pushed. But if that happened regularly, who could justify the code review process at all?

In real life, what often happens is this:

1. A code review is started.
2. A number of reviewers make several suggestions: some small and easy, some fluffy without obvious solutions, and some complicated.
3. The author makes some changes: at least the easy fixes or maybe several changes in an effort to make the reviewers happy. The author might ask the reviewers questions to clarify things, or the author might make comments to explain why particular changes weren’t made.
4. The reviewers come back, accept some of the updates, make further comments on others, find other things they don’t like about the original code, respond to questions, and argue with other reviewers or the author over their comments in the review.
5. The code author makes more changes, adds more comments and questions, and so on.
6. The reviewers check the changes, make more comments and suggestions, and so on.
7. Steps 5 and 6 are repeated, perhaps forever.

During the process, in theory, changes and comments should decline towards zero until the code is ready. The most depressing case is when each iteration brings at least as many new issues as old ones that were closed. In such a case, the team has entered the “infinite loop of code review.” This happens for a number of reasons:

- It happens if reviewers nitpick and if they give inconsistent feedback. For reviewers who fall into these habits, there’s an infinite number of problems to identify and an infinite number of comments to make.
- It happens when there’s no clear purpose to the review or no guidelines to follow during reviews, because every reviewer then feels every possible problem must be found.
- It happens when it’s unclear what a reviewer’s comments require from the code author. Does each comment imply a change that must be made? Are all questions an implication that the code isn’t self-documenting and needs to be improved? Or are some comments made simply to educate the code author for next time, and are questions posed just to help the reviewer understand and learn?
Comments should be understood either as showstoppers or not showstoppers, and if reviewers decide the code needs to change before they can sign off on it, they need to be explicit about exactly what actions the code author should take.

It’s also important to understand who is responsible for deciding the review is “done.” This can be achieved through a task list of items that have been checked and have passed, or it can be accomplished by an individual who is empowered to say, “good enough.” There usually needs to be someone who can break stalemates and resolve disagreements. This might be a senior developer, a lead, or an architect—or it might even be the code author in teams in which there is a high degree of trust. But at some point, someone needs to say either “the review is finished” or “when these steps are complete, the review is finished.”

**Antipattern: GhostReviewer**

Here’s where I admit the antipattern that I am most guilty of committing: ghosting. Whether I’m a reviewer or a code author, there comes a point in the code review (sometimes right at the start!) where I simply don’t respond during the review. Maybe there’s an important or interesting feature that I’ve been asked to review, so I decide to leave it until “a better time” when I can “really look at it properly.” Or maybe the review is large, and I want to set aside plenty of time. Or perhaps I’m the author, and after an iteration (or twenty) I just can’t face reading and answering the comments anymore, so I decide to wait “until my head is in the right place.”

Sound familiar?

Whatever the cause, sometimes someone in the review process simply doesn’t respond. This can mean the review is dead in the water until this person looks at the code. This is a waste: Even though someone has invested time in creating an asset (the new code), until it’s in production it’s not adding value. In fact, it’s probably rotting as it gets further and further behind the rest of the codebase.

Several factors can cause ghost reviews. Large code reviews are one factor, because who wants to wade through tens or hundreds of changed files? Not valuing code reviews as real work or part of the deliverables is another factor. Difficult or demoralizing code review experiences are another major factor: No one wants to stop coding (something developers generally enjoy) to participate in a time-consuming and soul-destroying activity.

Here are suggestions for addressing ghost reviews:

- Make sure code reviews are small. Each team has to work out its definition of this, but it’ll be in the region of hours or days of work to review, not weeks.
- Make sure the purpose of the code review is clear and what reviewers should be looking for is clear. It’s hard to motivate yourself to do something when the scope is “find any possible problem with the code.”
- Allow time in the development process for doing code reviews.

This last point might require team discipline, or the team might want to encourage allowing time by (for example) rewarding good code review behavior via objectives or whatever mechanism is used to determine a developer’s productivity.

**What Can Your Team Do?**
Focus on creating a solid code review process. I've written about it on my blog but would like to share part of that process here.

There are many things to consider when doing a code review, and if developers worried about all of them for every code review, it would be nearly impossible for any code to pass the review process. The best way to implement a code review process that works for everyone is to consider the following questions:

- Why is the team doing reviews? Reviewers have an easier job when there's a clearly defined purpose, and code authors will have fewer nasty surprises from the review process.
- What are team members looking for? When there's a purpose, developers can create a more focused set of things to check when reviewing code.
- Who is involved? Who does the reviews, who is responsible for resolving conflicts of opinion, and who ultimately decides if the code is good to go?
- When does the team do reviews, and when is a review complete? Reviews could happen iteratively while developers are working on the code or at the end of the process. A review could go on forever if there isn't clear guidance on when the code is finally good to go.
- Where does the team do reviews? Code reviews don't require a specific tool, so reviews could be as simple as authors walking a colleague through their code at their desk.

Once these questions are answered, your team should be able to create a code review process that works well. Remember, the goal of a review should be to get the code into production, not to prove how clever the developers are.

**Conclusion**

Code review antipatterns can be eliminated, or at least mitigated, by having a clear code review process. Many teams believe they should be doing code reviews, but they don’t have clear guidelines on why they're doing them.

Different teams need different types of code review, in the same way that different applications have different business and performance requirements. The first step is to figure out why the team needs to review code, and then the team can work on

- Automating easy checks (for example, checking code style, identifying common bugs, and finding security issues)
- Creating clear guidelines on when a review happens, what to look for, and who decides when reviews are finished
- Making code reviews a key part of the development process

Focusing on why code reviews are done will help teams create best practices for a code review process so it will be much easier to avoid code review antipatterns.
is a leader of the Sevilla Java User Group and a Java Champion. She believes healthy communities and sharing ideas help us to learn from mistakes and build on successes. As a Developer Advocate for JetBrains, she gets to share all the interesting things she’s constantly discovering. Follow her on Twitter at @trisha_gee.