10 tips for better Pull Requests by Mark Seemann

Making a good Pull Request involves more than writing good code.

The Pull Request model has turned out to be a great way to build software in teams - particularly for distributed teams; not only for open source development, but also in enterprises. Since some time around 2010, I've been reviewing Pull Requests both for my open source projects, but also as a team member for some of my customers, doing closed-source software, but still using the Pull Request work flow internally.

During all of that time, I've seen many great Pull Requests, and some that needed some work.

A good Pull Request involves more than just some code. In most cases, there's one or more reviewer(s) involved, who will have to review your Pull Request in order to evaluate whether it's a good fit for inclusion in the code base. Not only must you produce good code, but you must also cater to the person(s) doing the review.

Here's a list of tips to make your Pull Request better. It isn't exhaustive, but I think it addresses some of the more important aspects of creating a good Pull Request.

1. Make it small

A small, focused Pull Request gives you the best chance of having it accepted.

The first thing I do when I get a notification about a Pull Request is that look it over to get an idea about its size. It takes time to properly review a Pull Request, and in my experience, the time it takes is exponential to the size; the relationship certainly isn't linear.

If I get a big Pull Request for an open source project, I do realize that the submitter has most likely already put in substantial work in his or her spare time, so I do go to some lengths to review a big Pull Request, even if I think it's too big - particularly when it looks like it's a first-time contributor. Still, if the Pull Request is big, I'll need to *schedule* time to review it: I can't review a big chunk of code using five minutes here and five minutes there; I need contiguous time to do that. This already introduces a delay into the review process.

If I get a big Pull Request in a professional setting (i.e. where the submitter is being paid to write the code), I often reject the Pull Request simply because of the size of it.

How small is small enough? Obviously, it depends on what the Pull Request is about, but a Pull Request that touches less than a dozen files isn't too bad.

2. Do only one thing

Just as the Single Responsibility Principle (http://en.wikipedia.org/wiki/Single_responsibility_principle) states that a class should have only one responsibility, so should a Pull Request address only a single concern.

Imagine, as a counter-example, that you submit a Pull Request that addresses three independent, separate concerns (let's call them A, B, and C). The reviewer may immediately agree with you that A and C are valid concerns, and that your solution is correct. However, the reviewer has issues with your B concern. Perhaps he or she thinks it's not a concern at all, or she disagrees with the way you've addressed it.

This becomes the start of a lengthy discussion about concern B, and how it's being addressed. This discussion can go on for days (particularly if you're in different time zones), while you attempt to come to agreement; perhaps you'll need to make changes to your Pull Request to address the reviewer's concerns. This all takes time.

It may, in fact, take so much time that other commits have been merged into *master* in the meantime, and your Pull Request has fallen so much behind that it no longer can be automatically merged. Welcome to Merge Hell.

All that time, your perfectly acceptable solutions to the A and C concerns are sitting idly in your Pull Request, adding absolutely no value to the overall code base.

Instead, submit three independent Pull Requests that address respectively A, B, and C. If you do that, the reviewer who agrees with A and C will immediately accept two of those three Pull Requests. In this way, your non-controversial contributions can immediately add value to the code base.

The more concerns you address in a single Pull Request, the bigger the risk that at least one of them will block acceptance of your contribution. Do only one thing per Pull Request. It also helps you make each Pull Request smaller.

3. Watch your line width

The reviewer of your Pull Request will most likely be reviewing your contribution using a diff tool. Both GitHub (https://github.com) and Stash (https://www.atlassian.com/software/stash) provide browser-based diff views for reviewing. A reviewer can even configure the diff view to be side-by-side; it makes it much easier to understand what changes are included in the contribution, but it also means that the code must be readable on half a screen.

If you have wide lines, you force the reviewer to scroll horizontally.

There are many reasons to keep line width below 80 characters (http://richarddingwall.name/2008/05/31/is-the-80-character-line-limit-still-relevant); making your code easy to review just adds another reason to that list.

4. Avoid re-formatting

You may feel the urge to change the formatting of the existing code to fit 'your' style. Please abstain.

Every byte you change in the source code shows up in the diff views. Some diff viewers have options to ignore changes of white space, but even with this option on, there are limits to what those diff viewers can ignore. Particularly, they can't ignore if you move code around, so please don't do that.

If you really need to address white space issues, move code around within files, change formatting, or do other stylistic changes to the code, please do so in an isolated pull request that does only that, and state so in your Pull Request comment.

5. Make sure the code builds

Before submitting a Pull Request, build it on your own machine. True, *works on my machine* isn't particularly useful, but it's a minimum bar. If it *doesn't work on your machine*, it's unlikely to work on other machines as well.

Watch out for compiler warnings. They may not prevent you from compiling, so you may not notice them if you don't explicitly look for them. However, if your Pull Request causes (more) compiler warnings, a reviewer may reject it; I do.

If the project has a build script, try to run that, and only submit your pull request if the build succeeds. In many of my open source projects, I have a build script that (among other things) treats warnings as errors. Such a build script may automate or implement various rules for that particular code base. Use it before submitting, because the reviewer most likely will use it before merging your branch.

6. Make sure all tests pass

Assuming that the code base in question has automated tests, make sure all tests pass before submitting a Pull Request.

This should go without saying, but I regularly receive Pull Requests where one or more tests are failing.

7. Add tests

Again, assuming that the code in question already has automated (unit) tests, do add tests for the code you submit.

It doesn't often happen that I receive a Pull Request without tests, but when I do, I often reject it.

This isn't a hard rule. There are various cases where you may need to add code without test coverage (e.g. when adding a Humble Object (http://xunitpatterns.com/Humble%20Object.html)), but if it can be tested, it should be tested.

You'll need to follow the testing strategy already established for the code base in question.

8. Document your reasoning

Self-documenting code rarely is.

Yes, code comments are apologies (http://butunclebob.com/ArticleS.TimOttinger.ApologizeIncode), and I definitely prefer well-named operations, types, and values over comments. Still, when writing code, you often have to make decisions that aren't self-evident (particularly when dealing with Business 'Logic').

Document *why* you wrote the code in the way you did; not what it does.

My preferred priority is this:

1. **Self-documenting code:** You *can* make some decisions about the code self-documenting. Clean Code (http://amzn.to/XCJi9X) is literally a book on how to do that.
2. **Code comments:** If you can't make the code sufficiently self-documenting, add a code comment. At least, the comment is co-located with the code, so even in the unlikely event that you decide to change version control system, the comment is still preserved. Here's an example where I found a comment more appropriate than attempting to design my way out of the problem. (https://github.com/GreanTech/AtomEventStore/blob/e5fe679c08abb6fe108509117651d29dce17e270/AtomEventStore/AtomEventStorage.cs L71)
3. **Commit messages:** Most version control systems give you the opportunity to write a commit message. Most people don't bother putting anything other than a bare minimum into these, but you can document your reasoning here as well. Sometimes, you'll need to explain why you're doing things in a certain order. This doesn't fit well in code comments, but is a good fit for a commit message. As long as you keep using the same version control system, you preserve these commit messages, but they're once removed from the actual source code, and you may loose the messages if you change to another source control system. Here's an example where I felt the need to write an extensive commit message (https://github.com/GreanTech/AtomEventStore/commit/615cdee2c4d675d412e6669bcc0678655376c4d1), but I don't always do that (https://github.com/GreanTech/AtomEventStore/commit/e5fe679c08abb6fe108509117651d29dce17e270).
4. **Pull Request comments:** Rarely, you may find yourself in a situation where none of the above options are appropriate. In Pull Request management systems such as GitHub or Stash, you can also add custom messages to the Pull Request itself. This message is twice removed from the actual source code, and will only persist as long as you keep using the same host. If you move from e.g. CodePlex to GitHub, you'll loose those Pull Request messages. Still, occasionally, I find that I need to explain myself to the reviewer, but the explanation involves something external to the source code anyway. Here's an example where I found that a reasonable approach. (https://github.com/GreanTech/AtomEventStore/pull/70)

You don't need to explain the obvious, but do consider erring on the side of caution. What's obvious to you today may not be obvious to anyone else, or to you in three months.

9. Write well

Write good code, but also write good prose. This is partly subjective, but there are rules for both code and prose. Code has correctness rules: if you break them, it doesn't compile (or, for interpreted languages, it fails at run-time).

The same goes for the prose you may add: Code comments. Commit messages. Pull Request messages.

Please use correct spelling, grammar, and punctuation. If you don't, your prose is harder to understand, and your reviewer is a human being.

10. Avoid thrashing

Sometimes, a reviewer will point out various issues with your Pull Request, and you'll agree to address them.

This may cause you to add more commits to your Pull Request branch. There's nothing wrong with that per se. However, this *can* lead to unwarranted thrashing.

As an example, your pull request may contain five commits: A, B, C, D, and E. The reviewer doesn't like what you did in commits B and C, so she asks you to remove that code. Most people do that by checking out their pull request branch and deleting the offending code, adding yet another commit (F) to the commit list: [A, B, C, D, E, F]

Why should we have to merge a series of commits that first adds unwanted code, and then removes it again? It's just thrashing; it doesn't add any value.

Instead, remove the offending commits, and force push your modified branch: [A, D, E]. While under review, you're the sole owner of that branch, so you can modify and force push it all you want.

Another example of thrashing that I see a lot is when a Pull Request is becoming old (often due to lengthy discussions): in these cases, the author regularly merges his or her branch with *master* to keep the Pull Request branch up to date.

Again: why do I have to look at all those merge commits? You are the sole owner of that branch. Just rebase (http://git-scm.com/book/en/v2/Git-Branching-Rebasing) your Pull Request branch and force push it. The resulting commit history will be cleaner.

Summary

One or more persons will review your Pull Request. Don't make your reviewer work.

The more you make your reviewer work, the greater the risk is that your Pull Request will be rejected.

← Previous (/2015/01/10/diamond-kata-with-fscheck/) Archive (/archive)

Next → (/2015/01/19/from-primitive-obsession-to-domain-modelling/)

Wish to comment?

You can add a comment to this post by sending me a pull request (https://github.com/ploeh/ploeh.github.com#readme). Alternatively, you can discuss this post on Twitter or somewhere else with a permalink. Ping me with the link, and I may respond.

Published

Thursday, 15 January 2015 10:06:00 UTC

Tags

Productivity ^{27} (/tags#Productivity-ref)

Buy me a cup of coffee (https://www.paypal.com/cgi-bin/webscr?cmd=_s-xclick&hosted_button_id=NZEPYW8KVZ8WL)
Buy my book *Dependency Injection in .NET* (http://amzn.to/12p90MG)
Watch my Pluralsight courses (http://blog.ploeh.dk/pluralsight-courses)
Watch my Clean Coders videos (https://cleancoders.com/videos/humane-code-real)
Public speaking schedule (http://blog.ploeh.dk/schedule)

Tweet

"*Our team wholeheartedly endorses Mark. His expert service provides tremendous value.*"
Hire me! (http://blog.ploeh.dk/hire-me)