Good practices with version control systems

Damien Goutte-Gattat <dgouttegattat@incenp.org>

2024/06/03

Abstract

This document is intended to capture good practices for working with version control systems.

Table of Contents

1. Writing good commit messages

1.1. Why are good commit messages important?

There are actually two aspects to that question: the first is, “why documenting your changes is important?” and the second is, “why is it important to do that in commit messages?”

1.1.1. Why is documenting your changes important?

Because a change is made only once, but may need to be studied many times over.

Whenever you make some change to a project, at the time you are making the change, the reason why you are doing it is perfectly obvious to you. After all, you just spent the last couple of hours/days/weeks (depending on how unlucky you were) trying to make that exciting new feature work or fixing that goddamn bug – you know very well what you are doing and why.

Thing is, you are in fact the only person on Earth to know that. Nobody else has done your work implementing that feature or figuring out that bug. What is painstakingly obvious to you is likely to be a complete mystery to everybody else.

1.1.1.1. “OK, but once the change is done, it doesn’t matter anymore, right?”

Wrong. You cannot assume that now that your change is committed, that’s the last time anyone will ever have to worry about it. At any time in the future, someone may need to have a look at that change, for a variety of reasons. And that person will not know

what you currently know about what you did and why. This is true even if the person who needs to have a look at your change in the future happens to be none other than yourself: five years from now, will you remember why you did that change in the early morning of 1st June 2024?

1.1.1.2. “Still not convinced this is really useful. Sounds like a waste of time to me.”

Once you’ve taken the habit of always documenting your changes, most of the time it shouldn’t take more than a few minutes to compose a good message whenever you are ready to commit your changes. This is nothing compared to the time wasted by people (including your future self) who will need to understand your change in the future.

The author of those lines once lost an entire evening trying to figure out why a build suddenly started failing after pulling the latest changes from fellow contributors. It turned out that one of those changes was bogus, but that particular change was “hidden” in a commit whose message was simply “update Makefile” – lost among dozens of other commits with the same message. Just finding the offending commit took a lot of time, and once it was found, it was impossible to guess why the bogus change was made. It thus required going through the discussion of all the pull requests of the day to find a brief mention of the change.

Committing changes without documenting them sends to all your fellow contributors the signal that you don’t care about *their* time. You’d rather have them waste countless hours trying to understand why you did what you did, than spend a couple of minutes explaining it once and for all. It’s actually one of the most selfish acts you can do as a contributor to a free software project.

1.1.1.3. “Well, I am the only person working on this project anyway, so there’s no point.”

First, this is a disservice to your future self. You may very well need to come back to your changes in the future. Second, you may be the only contributor *now*. It doesn’t mean you will always be.

Fun fact: that you do not document your changes may actually be one of the reasons you are the only person working on your project. It is entirely possible that someone may have been interested in contributing at some point, may have had a look at your code, but realised that they couldn’t understand the rationale behind each change in the project’s history, and therefore decided to move on and to go contribute to a project that has better practices instead.

1.1.2. Why document changes in commit messages?

Because commit messages are the only things that are guaranteed to exist for the lifetime of the project, regardless of where the project is hosted and how it is managed.

Commit messages are stored directly within the version control system (VCS, e.g. Git, Mercurial, etc.) that the project is using. Whenever you clone a repository on your local machine, you automatically get all the commits that make up the history of the project (or, at least, the history of the master branch), along with all the associated commit messages. This is true regardless of the repository hosting service (e.g. GitHub, GitLab, SourceForge, etc.) that is used to host the repository. This means that: (1) those commit messages will always exist for as long as even a single copy of the repository exists somewhere; (2) those commit messages are always available to you at anytime, even if you happen to be working “offline” for any reason (for example, when aboard a train). There-

fore, commit messages are the place of choice to record anything that is worth knowing about the changes made to a project.

Bug tracker tickets, pull requests, wikis, forums, or any similar features that may be offered by the repository hosting service are, by contrast: (1) only available if you are online; (2) tied to the hosting service. If the project decides to move from one hosting service provider to another, it may be possible to transfer bug tracker tickets or similar items to the new provider, but this is absolutely not guaranteed; this is entirely dependent on the good will of both the provider that the project is leaving and the provider that the project is moving to. Even if the project you are contributing to currently has no intention of ever changing its hosting service provider, it may very well end up not having a choice. Providers come and go and just because a given provider has existed for a long time, it does not mean that it will always exist and/or that it will always be willing to host your project (especially if, as in many cases, you do not actually pay for the service). In short, whatever happens in a bug tracker ticket should always be considered *volatile* – something that could disappear anytime or not be available, even temporarily, whenever you need it. Good for discussing an issue (e.g. ponder the pros and cons of different possible solutions to the issue, ironing out the details on how to implement a requested feature, etc.), but not a good place to store long-term information.

1.1.3. Why not comment the code?

Comments embedded directly within the code are also a good place to record long-term information about the code. Because they are inside the files that are under version control, they also exist for the lifetime of the project.

But code comments and commit messages serve slightly different purposes. Code comments are good to explain non-obvious details about what the current code is doing; commit messages are good to explain non-obvious details about how the current code came to be. Or to put it differently, comments are for documenting the code; commit messages are for documenting the code *changes*. They are not mutually exclusive, in fact they are complementary.

1.1.4. What about the project's documentation?

This is similar to the point above regarding code comments. If your project has some documentation files (it probably should, by the way!), and those files are under version control in the project's repository, then they are also a good place to record long-term information.

But similarly, the project's documentation and the commit log history serve different purposes. The documentation is here to explain whatever needs to be explained about the current state of the project; the commit log history is about documenting how the project got to its current state.

You do not want to clutter your current documentation with details on all the changes that occurred since the project was started years ago – for most readers of the documentation (e.g. your users, or fellow contributors if we're talking about technical documentation), such historical details are likely to be irrelevant. They will only be relevant when people will need, for one reason or another, to dig into the history to understand why a particular change was made.

1.2. What is a good commit message?

A good commit message is a message that explains: (1) what you are trying to do in the commit, and (2) why you are doing it.

Explaining what you are trying to do is important for two reasons. First, it gives whoever is reading your commit a chance to check whether your actual change (as shown by the "diff" of the commit itself) indeed does what it was supposed to do. Basically, the commit message is "here is what I wanted to do", and the diff is "here is how I did it". If you don't explain what the commit is supposed to do, the reader will have to figure that out by reading the diff, but then they will have no way of knowing whether what the commit actually does is really what it was meant to do. This is especially important in the context of a pull request review – when the person who is reading your commit message is the reviewer who will decide whether to approve your PR.

Second, it provides *context* that may not be available in the diff itself, but that may be useful to understand the overall commit. Keep in mind that the reader may not have the entire codebase in their head (this is true even if the reader happens to be the chief maintainer of the project you are contributing to – for any large enough project, *no one* can be familiar enough with the entire codebase that they will instantly recognise the lines affected by the diff and immediately know what the change is all about).

Explaining why you are doing what you do is important because that's something that is typically not inferrable at all from the diff. Are you fixing a bug? What was the bug? Are you improving something? What are you improving? Speed? Memory consumption? Are you implementing a new feature? What feature is it? Is it a feature that was requested specifically by someone? Are you scratching one of your itches?

1.2.1. Referring to a bug tracker ticket is not enough

If a bug has been described in a bug tracker ticket (or, if a new feature has been requested in a ticket), it is of course useful to refer to the corresponding ticket in a commit that fixes the bug (or implements the requested feature). But the reference to the ticket cannot be the entirety of the commit message.

The commit message should be *self-sufficient*. If the commit fixes a bug, its message should expose briefly: (1) what the reported bug was, (2) what the cause of the bug was, and (3) what the fix is (not necessarily in that order). If the commit implements a new feature, its message should expose briefly what was the requested feature.

This doesn't mean that the commit message has to repeat everything that is said in the bug tracker ticket. If there was a long discussion on the ticket about, say, the putative causes of the bug, the best way to fix it, or the pros and cons of implementing the requested feature, it may be fine for the commit message to include something like, "see the linked ticket for more details, in particular for why we didn't choose the more obvious route of doing bla bla bla". That should be done sparingly, though.

1.2.2. Do not paraphrase the diff

When explaining the "what", give an overview of the change, not a line-by-line explanation of it. It is safe to assume that whoever will read your commit will be reasonably familiar with whatever language or technology is used by the project. Therefore, they don't need a step-by-step explanation of what your change does. (If they are not yet familiar with the language or technology, well, they will have to learn, but commit messages are not the correct medium to teach them.)

Explain what is not obvious from reading the diff. This may include details about stuff that are outside the window of the diff. For example, if you modify a particular function, the diff will likely not show the part(s) of the code where the function is called – so it may be useful to briefly state what the function is about and when it is called. Remember: you know that, but your readers (including your future self) may not. Make their life easy by dispensing them from having to dig into the codebase to find all the callers of the function you changed.

1.2.3. Explain as much as needed – nothing more, nothing less

There is no “correct” length for a commit message. Some changes may require more than 50 lines to be fully explained, while some other changes will need no more than one sentence of 2 or 3 lines. Exercise your judgement, and do not try to be excessively verbose to “fill” a message that otherwise would seem too short to you, or conversely to be excessively concise (losing some precious bits of information in the process) to shorten a message that otherwise would seem too long.

Sometimes, what a commit does and even why it does may be entirely evident from the diff itself, in which case there may be no need to elaborate in the commit message. Typical examples are documentation changes or typo fixes. In such cases it is OK for a commit message to be composed of only the header line (which may simply be “documentation update” or “typo fix”), without an actual body.

Note that even when a change is evident from the diff, there may still be things worth mentioning in the commit message. For example, let’s say you reorganised an entire chapter of a documentation (moving sections and paragraph around to make the text flow better): it may be useful to explicitly state that you merely reorganised the contents of the chapter without introducing meaningful changes in the contents (e.g. you didn’t add or remove any section). This will let your fellow contributors know immediately what to expect, without forcing them to scan the entire diff to realise that, “oh, OK, apparently they just moved things around here”.

A particular case is when you did some kind of “automated” change – for example, you used a tool to automatically apply some standard formatting rules to an entire file (or maybe even the entire codebase); in that case, it is advised to include in the commit message the precise command used to invoke the tool that performed the change.

1.2.4. Feelings or doubts are not out-of-place

Commit messages don’t have to be solely about facts. If you are feeling unsure about the change you are making, or you don’t like some aspects of it, you may include that in your commit message. For example, if you fixed a bug but are concerned about possible performance issues as a result of your fix, and you believe that a better solution should be found if possible, that is clearly something that would be valuable to other contributors, and it should absolutely be part of the record.

This includes criticism of the existing code (rather than the code you are introducing yourself), but be careful in that case to really criticise the *code*, not the person(s) who wrote it in the first place (even if that person is yourself). “The fix proposed here works for now, but I believe this entire function is suboptimal and we should devise a better approach at some point” is fine; “whoever wrote this function has clearly no idea on how to efficiently use a hash table” is not.

1.3. What is the form of a good commit message?

Many guidelines about how to write good commit messages focus on the form, with recommendations such as “capitalise the first word”, “use imperative mood”, or “specify the type of commit [with the use of agreed upon keywords]” and so on.

Such guidelines completely miss the point of what a good commit message is about.

What matters in a commit message is the *contents*, not the *form*. A badly formatted message that explains what you did and why you did it will always be better than a perfectly formatted message that does not explain anything useful.

The only case where you should worry about the form is if the project you are contributing to happens to have its own guidelines mandating a specific format for the commit

messages – in which case you should obviously try to follow those guidelines as much as possible. Many projects don't have specific guidelines though, and in their absence you should feel free to write your messages in whatever way you like – just be sure to explain what needs to be explained.

1.3.1. The header line

The one exception to the "form does not matter" rule is that your message *must* always start by a (preferably short) one-line summary (hereafter called the "header line"), followed by a blank line, followed by the body of the commit message. This is because most if not all tools assume the presence of such a header line.

It is often difficult to know what to put in the header. A change may very well be complex enough that summarising it in one (short) line is nearly impossible. Do your best, focusing on the "what" part. Keep in mind that many tools, when listing commits, will by default only show the header line of each commit. Put yourself in the shoes of someone looking for a particular commit, and try to write a header line that will make it easy for that person to figure out whether your commit might be the commit that they are looking for, without having to display the full message for each commit.

Of note, because the header is intended as a summary, it is perfectly OK to repeat yourself between the header and the body.

1.3.2. Capitalisation

It doesn't matter at all. A commit message is not a good commit message because it starts with a capital letter and ends with a final dot.

That being said, from the point of view of the reader, a message with properly capitalised and punctuated sentences is nicer to read.

1.3.3. Grammar

Do *not* worry about grammar. As long as your message is legible and understandable, nobody will ever blame you if it contains a grammar error or a typo.

1.3.4. Style

Just use whatever style you want (unless, again, you're contributing to a project that mandates a given style). You can use a direct, first-person account ("I noticed the variable foo was not properly initialised, and that it could lead to a NullPointerException in some specific conditions. So here I initialise the variable in the constructor of Bar."). If you are not comfortable using the first person, you can use the passive voice instead ("a proper initialisation procedure was added to ensure that no variables could ever be used without being initialised"). Or you can use the imperative mood ("add a proper initialisation procedure for foo"), or some noun forms ("addition of a proper initialisation procedure"). Or a mix of all that.

For what it's worth, the author of those lines tends to favour the first person plural ("we introduce a new option to allow bla bla bla"), even when he is the only person writing the commit – that is known in some circles as the "royal we", and is also fine. (He also likes speaking about himself in the third person, apparently.)

Similarly, there is no inherently "good" order for how you explain your commit. You can start by first describing "why" a change was needed (what was the problem in the existing code), and then describe "what" the change is. Or you can do the opposite ("we do bla bla bla. This is because bla bla"). Whatever suits you.

You are allowed to be creative. Feel free to write a commit message as a series of haikus if it would amuse you. As long as you explain what you did and why.

1.3.5. Use of keywords

You may use keywords in your commit message (especially in the header line) to give an immediate information about the type of change you are making (e.g., `bugfix` for a change that, well, fixes a bug, `doc` for a change that improves the documentation, `build` for a change that affects the build system, etc.).

This is not required, however. If you did a good job explaining your change, it should be obvious what the type of the change is, without having to refer to a keyword.

The exception is for keywords that link a commit to an issue in the bug tracker of the project. If you want to say that your commit fixes the issue number 123, it's better to use the `closes` keyword, followed by the issue number, on a separate line, rather than writing a full sentence like "This commit fixes the issue 123". Most repository hosting services will recognise such a keyword.

1.4. How to make sure your project has good commit messages

This section is mostly targeted at the *maintainers* of a project.

1.4.1. Do not implement an automated check at commit time

Most version control systems, such as Git, can be configured to run some scripts automatically whenever a change is committed, to perform some kind of checks on the change. You can find several examples on the Internet of scripts intended to check whether the commit message follows some predefined rules – for example, it has proper capitalisation and punctuation, the header line includes a keyword, it has a minimum length of X lines, etc.

As discussed in the previous section, this completely misses the point of what a good commit message is, by focusing on the form rather than the contents. Using such a script will only teach the persons who contribute to your project how to blindly follow some meaningless formatting rules just to make your script happy, instead of teaching them how to *think* about what they need to put in a commit message.

1.4.2. Make sure that you write good commit messages

You cannot ask your fellow contributors to write good commit messages if you, and all the regular maintainers involved in your project, don't already do it yourself. For two reasons:

First, it makes you look like a damn hypocrite, and/or some gatekeeper trying to actually prevent people from contributing by applying to them some rules that don't apply to you or the regular maintainers.

Second, people need to see what good commit messages look like. It is quite possible that they have never learned anywhere how to write good commit messages, so they need to be led by example. You must be able to show them some commits from the history of your project and tell them, "this is what I mean when I ask you to write good commit messages".

1.4.3. Reject PRs that have bad commit messages

When reviewing a pull request, do not only check the changes themselves. Check whether the changes are correctly documented in every commit message. If they are not, then

close the PR and ask the contributor to submit another one with better messages. Yes, it can be hard to do – the PR may otherwise be a good PR, with changes that you are actually keen to merge, and you don't want to discourage contributors. But if you want to ensure that your project's history is properly documented, that is the price to pay.

Many free software projects have such a policy, and their contributors are perfectly able to understand it and comply with it.

In some circumstances, you may want to redo the PR yourself, by cherry picking each commit one by one on a separate branch and amending their messages. You could then show your version of the PR to the original contributor and say something like, "this is what I would have wanted you to do; I've done it for this time, but please make sure that your next PR doesn't need it".

Another possibility, if the entire PR is small enough, is to do a "squash merge", where all the commits that make up the PR are merged to the target branch as a single commit, with a message that you can write at merge time. Don't do that if the PR contains several logical changes, though – all those changes would end up in a single commit, which is not good.

2. Managing Git history

2.1. Maintaining a "clean history"

2.1.1. What is a clean history?

To understand what is meant by "clean history", let's imagine the following scenario:

- Bob submits a pull request to Alice's project. The PR is made of, say, three distinct commits, that conjointly implements a new feature.
- As soon as the PR is submitted, a test suite automatically runs on his changes, and fails. It turns out there was a typo in the second commit. (Bob should probably have run the test suite locally even before submitting the PR.)
- Bob appends a fourth commit to the PR, in which he fixes the typo. The test suite now passes. Bob requests a review from Alice.
- Alice likes the new feature, but is not entirely happy with the way it has been implemented. She asks Bob to make some changes.
- Bob agrees with the requested changes, and appends two more commits (commit #5 and #6) to implement them.
- Alice is now satisfied with the current state of the PR. Charlie, a long-time contributor to the project, weighs in to suggest that the PR should also include some tests (to be added to the suite suite) for the new feature. Alice was initially ready to merge the PR, but upon seeing Charlie's suggestion she actually agrees that it would be a good idea to add some tests immediately.
- Bob complies and appends a 7th commit that adds the requested tests.
- By then, there have been some unrelated changes in the master branch of the project (made by Alice and other contributors), so Bob merges the master branch into the branch of his PR, creating an 8th commit. Luckily, there was no conflict.
- Both Alice and Charlie appreciate the new tests, but Alice points out that a particular condition is not tested, and she requests the addition of another test.

- Bob appends commit #9 to his PR, adding the requested supplementary test.
- Almost there. Alice is happy with the new feature, the way it is implemented, and the way it is tested. All that is missing now is a bit of documentation for the new feature, so she requests that.
- Bob appends commit #10 which adds a new section in the project's documentation to present and explain the feature.
- Everyone is now satisfied with the PR, so both Alice and Charlie approve it and Alice is ready to merge it.

Overall, this is a fairly common story, the likes of which happen everyday in the free software world.

At this point, just before being merged, Bob's PR contains 10 commits:

- commits #1 to #6 actually implement the new feature (#1, #2, and #3 represent the first attempt at implementing it, #4 is the typo fix, and #5 and #6 are the changes requested by Alice);
- commits #7 and #9 add the tests for the new feature;
- commit #10 adds the documentation;
- commit #8 is the merge commit that was needed to synchronise the PR with the master branch.

If Alice merges the PR as it is, all those 10 commits will be added to the tip of the master branch, along with an 11th commit that represents the merge operation itself.

The idea behind the notion of a "clean history" is that the definitive history of the project's changes (represented by the history of the master branch) should not contain all those commits, since they represent "trials and errors" steps that ultimately "pollute" the history. An "idealised" version of the history should only contain:

- a handful of commits implementing the new feature itself, without the initial implementation that Alice didn't like and without the typo (a mix of commits #1 to #6) – possibly even only one commit if the feature is simple enough that there is no need to break it down into several commits;
- one commit to add the tests (a mix of commits #7 and #9);
- one commit to add the documentation (commit #10).

2.1.2. Should a project have a clean history?

There is no consensus on that question, and this guide does *not* take a firm position.

Proponents of the "clean history" idea posit that the *readability* of the project's history is paramount, and that including in it "intermediate" changes that ultimately didn't make the cut (such as commits #1, #2, and #3 in the example above), changes that were only necessary because of the time it took for the PR to be approved (merge commit #8), or changes that are split over several commits for no good reason (commits #7 and #9 – the second one adds a test that should really have been added already in #7), makes the history needlessly more complicated, and therefore less useful, than it should be.

By contrast, opponents to the "clean history" idea posit that the *integrity* of the project's history is paramount, and that the history should merely give an accurate account of what really happened. Software development is messy, trials and errors are part of the

process, and it is perfectly normal for the project's history to reflect that. Some even argue that attempts to clean the history are dishonest and are intended to make the project's developers look smarter than they actually are, by pretending that there is never any "hiccup" in the project's development.

The two views are fundamentally incompatible, so it's up to each project to decide, in agreement with their regular contributors, whether they want to try maintaining a clean history or not.

A couple of objective points, however:

- Maintaining a clean history requires efforts and commitments on the part of everybody involved in the project, as well as a decent mastery of the version control system used by the project; that point alone might make adopting the "clean history" approach unrealistic for many projects.
- On the other hand, the "clean history" approach is successfully used by many free software projects, the most notorious of which being the Linux kernel.

2.1.3. How to maintain a clean history

If you decide that your project should have a clean history, there is really only one thing for you to do: you must only accept PRs that only contain the commits you want to see in your history.

That is, when a PR has been through some cycles of reviews, and as a result contains commits that "correct" previous commits in the same PR, you must not merge that PR. Instead, once the reviews are done and the PR has been brought to a state that you are happy with (all corrections implemented, all comments addressed), you must close that PR and ask its author to submit another PR – one that will, from the start, take into account all the comments made on the original PR.

You may have to explain tactfully to the author why submitting another PR is necessary, but usually, based on the experience of projects that have a "clean history policy" in place, the request to submit a clean PR is well received by contributors.

2.1.3.1. About squash merges

In some cases, when a PR is not fit to be merged into the master branch because it has a messy history, there is an alternative to requesting that a new PR be submitted: you may *squash* the PR instead. Squashing will blend all the commits of the PR into a single commit (whose commit message you can edit, so as to ensure it is a good commit message) that will then be appended to the master branch. One consequence of this process is that all the trials and errors of the PRs, all the initial attempts that had been corrected in later commits, will be "erased", and only the latest version of the PR will remain, thereby ensuring a "clean history" for the overall project.

It's an easy method to get a clean history from a messy PR, but it should only be used for simple PRs – when it is fine for all the changes made by the PR to be compiled into a single commit. In the example given above, Bob's PR is not eligible for that method, because even though it implements a single feature, it actually contains several distinct logical changes (the changes that implement the new feature itself, the changes to the test suite, and the changes to the documentation), that should not be lumped together in a single commit.