Tips for merge requests and code reviews

...or how to manage the process like a pro

Petr Zemek

October 11, 2019

TI Systems Seminar @ Avast

Outline

Introduction

Let's start with a bit of a discussion...

- What is the purpose of merge requests (MRs) and code reviews (CRs)?
- What do you like about MRs and CRs? What do you dislike? Or even hate?
- Why does GitHub call merge requests *pull requests* (PRs)?

How to create merge requests

Do a self-review before submitting the MR

The reviewer is not responsible for your carelessness.

The more complicated the MR the more detailed the description should be

Try to put yourself into the reviewer's shoes.

Describe:

- What were the major issues?
- Why did you decide to solve them in this way?
- Were there any other options?
- How do the commits relate to each other?
- Are there any problems to discuss?
- Include any relevant tickets from your bug-tracking system.

Make commits in MR read like a story

Who says that programmers cannot be writers?

Moreover:

- Commit-wise, the MR should be reviewable from the bottom to the top.
- Every commit should be atomic.

Do not be afraid to leave comments by yourself

If you want to discuss something with the reviewer, leave a comment.

Larger changes/MRs should be pre-approved before opening a MR

To minimize the risk of them not being accepted.

When in doubt, ask for a concept review to verify that the way you have decided to go has potential.

Every comment from the reviewer should make you think

- Why have I not thought about that?
- How can I improve the code/MR/... in the future?

Include only directly related changes

- Do not include irrelevant fixes of typos, formatting, etc.
- Generally, do not solve multiple issues in the same MR.

Squash minor fixes via interactive rebase

Do not include commits like *"Fix typo in previous comment"*.

Accept the fact that not all MRs will be merged

C'est la vie.

How to review merge requests

What to focus on (P1)

Does the code do what it should, nothing is missing, and does not it do something it should not do?

Additionally:

- Does the project function correctly and do all the tests pass?
- Are there tests for the new code?
- Has the documentation been updated?
- What about backward compatibility (versioning)?

What to focus on (P2)

Is the code safe?

- Are errors correctly handled?
- Is it impossible for the program to crash?
- Is the code free of security flaws?
- Is the code thread-safe?
- Are there no resource leaks?

What to focus on (P3)

Is the code readable, maintainable, and not needlessly inefficient?

- Does the code fit into the project or was it hacked there (e.g. shotgun surgery)?
- Is there a more idiomatic way of writing something?
- Can the code be shortened by using existing libraries?
- Is there no duplication?
- Are there no useless things that unnecessarily slow down the code?

What to focus on (P4)

Does the code conform to project's coding conventions?

- Spaces vs tabs.
- No useless trailing whitespace.
- Naming of variables (`snake_case` vs `camelCase`).
- Code formatting in general (placement of curly braces, line wrapping, etc.).
- Typos and grammar in strings/comments.

Be respectful, but brutally honest

If there is something wrong, it is your duty to report it, but in a respectful way.

You are reviewing the code, not the person

So let's not get personal.

Strive to make useful and informative remarks

And leave the useless ones at home...

- Include a reason *why*.
- If you criticize something, include an alternative way to consider.
- Include links to supportive material (articles, talks).
- Ask questions if you do not understand something.
- Ask questions to make the MR creator think (*"What happens if..."*).
- Report issues properly (steps to reproduce, expected behavior, actual behavior).
- Consider reporting an issue by crafting a failing test.

Show honest appreciation

Has to be honest and specific (i.e. not generic).

Examples:

- *"Cool, I did not know about distutils.util.strtobool(). Nice!"*
- *"Thank you for analyzing the Perl code, it must have been hard."*
- *"I have learnt a new word today ('spuriously'), thanks!"*

Always leave a comment

Even if only a plain and simple *"LGTM* 👍*"*.

Mind the wording

Pay close attention at the words that you choose.

- Use *"I suggest"* or *"Consider"* for non-critical issues.
- Use *we* instead of *I/you*.
- Prefix minor issues with *"Nitpick"*.

One MR can be reviewed by multiple people

Changes to critical parts of the code should be reviewed by multiple people.

Finish the review in a timely manner

Do not wait a month to do the review.

How to discuss comments

Show appreciation

- *"A very good point."*
- *"Nice catch!"*
- *"I did not know about that, thank you!"*
- *"The proposed alternative is indeed better. Let's use it."*

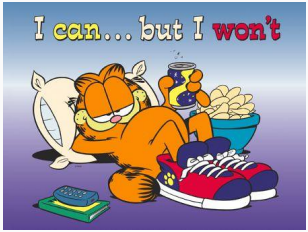
Do not take comments personally

It is (well, should be) the code that is being discussed, not you.

Do not be afraid to disagree

Code review should be a discussion, not a list of commands.

- However, if you disagree, you have to explain *why*.
- Please, let the reason not be *"Screw it, I am too lazy to do that"*.



https://www.flickr.com/photos/72665859@N03/6558098435

Do not be afraid to ask for help

You can tag (invite) other people and ask for their opinion.

Add a reaction to all comments and mark discussions as resolved

- Explain how the issue has been resolved.
- For trivial issues, marking the discussion as resolved is enough.

Summary

Let's summarize...

- Do a self-review before submitting the MR.
- Try to make the MR as reviewable as possible.
- Every comment from the reviewer should make you think.
- Focus on the most important things first.
- Strive to make useful and informative comments.
- Focus on the code, leave personal issues behind.
- Show honest appreciation.
- Do not be afraid to disagree.