Pull Requests, Code Reviews, and High-Quality Code

Petr Zemek

Lead Software Engineer at Avast
Threat Labs (Viruslab)
petr.zemek@avast.com
https://petrzemek.net

Practical Aspects of Software Design, BUT FIT, 2020-04-28

A tale of two workflows

The "lone wolf" workflow:

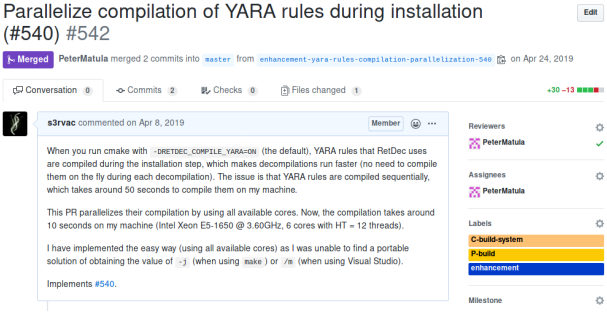
1. Put all your changes directly into `master`.
   (There is no step 2)

A more cautious workflow:

1. Create a new branch from the current `master`.
2. Implement the needed change there.
3. Push the branch and create a *pull request* (PR) from it.
4. Make the PR pass through a *code review* (CR).
5. The PR is approved and the branch is merged into `master`.

What is a pull request (PR)?

- A request to review your changes and merge them.
- Most commonly associated with PRs on GitHub:

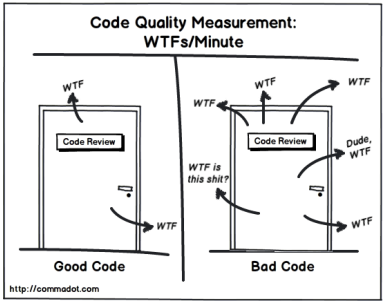


https://github.com/avast/retdec/pull/542

- Note: Called a *merge request* (MR) in some systems.

What is a code review (CR)?

- A process of looking at another person's code and verifying it is correct.
- Consists of:
  1. Writing comments towards the code.
  2. Giving evaluation (approve or request changes).
  3. Discussing comments with the author.



Reasons for creating PRs and doing CRs

- Finding bugs and other defects.
- Learning something new.
- Getting familiar with code that is new to you.
- Increasing the sense of mutual responsibility within your team.
- Finding a better solution.
- Slowing down the process of gradual degradation.
- Complying with formal requirements (e.g. QA).
- Running automated checks before the code is merged.
- Way of contributing to open-source projects.
- Make it harder for adversaries to sneak malicious code into the project.
- Writing better code.

Outline of the talk

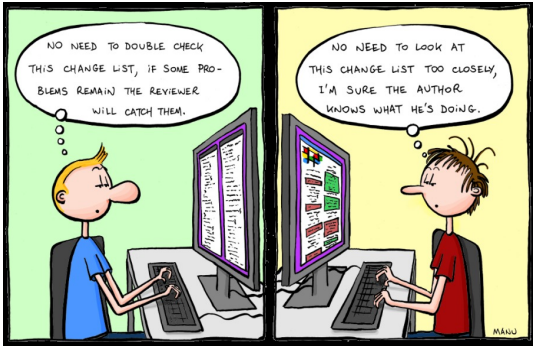
1. How to create PRs?
2. How to review PRs?
3. How to discuss comments?

Note: Writing PRs and doing CRs is a skill.

How to create PRs?

Do a self-review before submitting a PR

The reviewer is not responsible for your carelessness.



https://www.bebee.com/producer/@ebenezar-john-paul/code-review-checklist

More complicated PR ⇒ more detailed description

Try to put yourself into the reviewer's shoes.

Describe:

- What does the PR implement and why?
- What were the major issues?
- Why did you decide to solve them in this way?
- Were there any other options?
- Are there any problems to discuss?
- Include any relevant tickets from your bug-tracking system.
- Include *before* and *after* screenshots (if applicable).

Examples

https://github.com/rails/rails/pull/32865
https://github.com/discourse/discourse/pull/1320

Make atomic commits

A PR should be composed of *atomic commits*:

- Revolve around a single topic and one topic only.
- Separate refactoring, adding new functionality, and fixes.
- Are easy to reason about, review, revert, and bisect.

How to do them:

- Commit diligently/prudently.
- Use Git's *patch mode* (https://blog.petrzemek.net/2016/07/10/git-patch-mode-all-the-way/).
- Use Git's *interactive rebasing* (https://git-scm.com/book/en/v2/Git-Tools-Rewriting-History).

Smells:

- Use of *and* in commit messages.
- *"Screw it. Nobody will notice."*

Do not be afraid to leave comments by yourself

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

Larger changes/PRs should be pre-approved

- To minimize the risk of the changes/PRs not being accepted.
- When in doubt, ask for a concept review.

Every comment from the reviewer should make you think

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

Accept the fact that not all PRs will get merged

C'est la vie.

How to review PRs?

What to focus on (P1)

Does the code do what it should, nothing is missing, and does not 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?
- Isn't the implemented solution over-engineered?

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 PR 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 *"Looks good to me 👍"*.

https://knowyourmeme.com/photos/1287705-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 PR 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.

A maintainer's fail

https://github.com/JetBrains/teamcity-messages/pull/226

Anti-patterns

- Focusing primarily on nitpics
- Forcing subjective changes
- Forcing external contributors to fix nitpicks
- Inconsistent feedback
- "Bikeshedding"
- Back and forth (ping-pong) reviews
- Constantly bothering/interrupting the author during 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."*

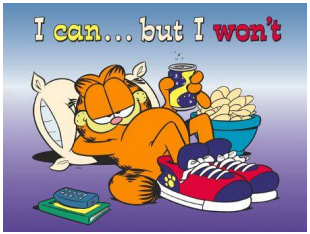
Do not take comments personally

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

How not to do it ;-)

https://github.com/pypa/twine/issues/153

Do not be afraid to ask for help

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

React 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 though.

Conclusion

What we have skipped

- How to select who should review the PR?
- How to review a large PR?
- GitHub/GitLab/BitBucket/... specifics
- Continuous integration (CI)
- PR hooks
- Bots
- Licensing, contributor license agreements (CLAs)
- ...

Recommended reading

- Andrew Hunt, David Thomas: The Pragmatic Programmer, Addison-Wesley, 1999
  In Czech: Andrew Hunt, David Thomas: Programátor pragmatik, Computer Press, 2007
- Steve McConnell: Code Complete (2nd edition), Microsoft Press, 2004
  In Czech: Steve McConnell: Dokonalý kód, Computer Press, 2006
- Robert C. Martin: Clean Code, Prentice Hall, 2008
  In Czech: Robert C. Martin: Čistý kód, Computer Press, 2009
- Robert C. Martin: The Clean Coder, Prentice Hall, 2011
  In Czech: -
- Sverre H. Huseby: Innocent Code, John Wiley & Sons, 2004
  In Czech: Sverre H. Huseby: Zranitelný kód, Computer Press, 2006

A bit of harmless self-promotion (my blog posts)

- Petr Zemek: Čistý kód, který funguje (2009-10-24)
- Petr Zemek: Vysoce kvalitní kód (2014-04-18)
- Petr Zemek: Důvody, proč psát jednotkové testy (2014-06-20)
- Petr Zemek: Zakomentovaný kód (2014-11-02)
- Petr Zemek: Udržitelný vývoj (2015-03-15)
- Petr Zemek: Proč rozlišovat jednotkové a integrační testy (2015-04-18)
- Petr Zemek: Na co se soustředit při revizích kódu (2018-05-08)
- Petr Zemek: Proč vytvářet funkce (2019-07-27)
- Petr Zemek: Tips for Creating Merge Requests and Doing Code Reviews (2020-01-31)
- Petr Zemek: Série "Chyby v návrhu"

Summary

- Strive to use workflows that utilize PRs and CRs.
- PRs and CRs provide many benefits.
- Writing PRs and doing CRs is a skill.
- Do a self-review before submitting a PR.
- Try to make the PR as reviewable as possible.
- Make atomic commits and atomic PRs.
- Every comment from the reviewer should make you think.
- When reviewing code, 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.