Code Review: Why and How

Living with "the invisible methods section"

Andy Boughton
Center for Statistical Genetics
October 2020

Why Code Review: Project Benefits

- Many research articles depend on methods implemented via code
- In addition to implementing equations, code must make many decisions (missing / imputed data, sample selection, underflow, etc)
- Reading the methods section is not enough to validate the program
  - ...but a code bug *is* enough to invalidate the conclusions of the paper
- Code review helps us to do good science today

Soergel, D. A. W. Rampant software errors may undermine scientific results. *F1000Res* **3**, 303 (2015). (link)
Chang, G. *et al.* Retraction. *Science* **314**, 1875.2-1875 (2006). (link)
Jay, C. *et al.* The challenges of theory-software translation. *F1000Res* **9**, 1192 (2020). (link)

Why Code Review: Developer Benefits

- Share new ideas and techniques so that developers aren't solving the same problems in isolation
- Reduce "bus factor": share critical information so that key pieces can be found, run, and maintained by other people
- Software and data are complex: catch subtle bugs/ edge cases
- Code review helps us to do good science tomorrow

Kemerer, C. F. & Paulk, M. C. The Impact of Design and Code Reviews on Software Quality: An Empirical Study Based on PSP Data. *IIEEE Trans. Software Eng.* **35**, 534–550 (2009). (link)

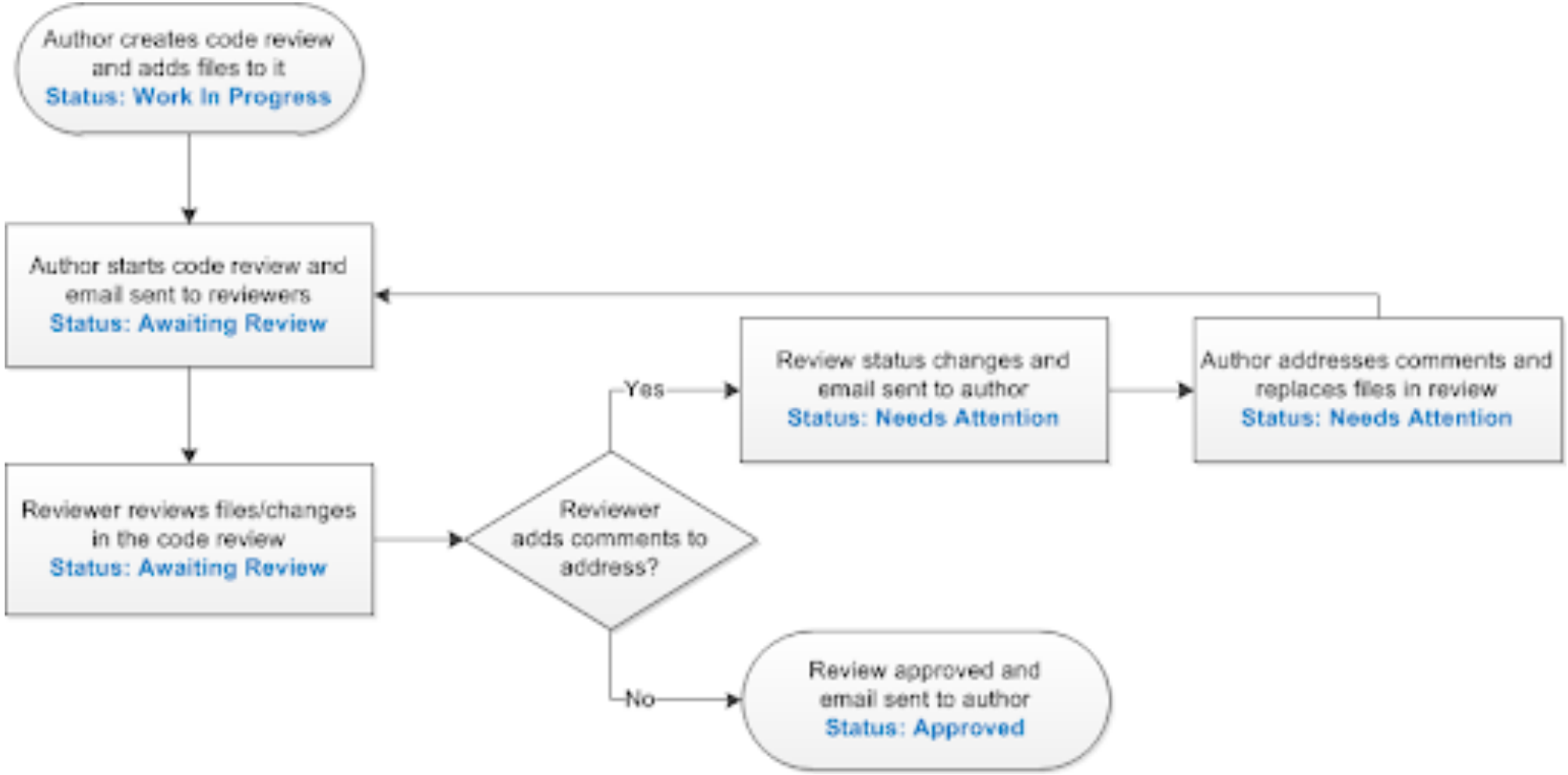
Credit: Disney/ Pixar, *Toy Story*. 1995

How to pick a good reviewer

- Code review is peer review!
- Should understand the basic language and tools
- Sufficient domain knowledge to determine intent of the code
- Code review requires practice: everyone should take turns

Petre, M. & Wilson, G. Code Review For and By Scientists. *arXiv:1407.5648 [cs]* (2014). (link)

Code review workflow



Credit: http://help.seapine.com/surround/2013.2.0/scmclient/Content/UserGuide/AboutCodeReviewProcess.htm

The role of the Reviewer

Prepare before commenting

- Wait until the code is ready (not “draft” / “work in progress”)
  - If automated checks fail, send it back
- Read the pull request description to understand the feature + any required setup
- Fetch the branch and make sure it runs.
  - Are the required input files available?
  - Do docs and build/deployment scripts work as written?
  - Does the program produce the expected output?

Strategies for reading code:
https://selftaughtcoders.com/how-to-quickly-and-effectively-read-other-peoples-code/

Give clear feedback

- Make comments targeted and actionable:
  - Instead of "everything is confusing", say "what if we tried x?"
  - Instead of "50 errors reported by tool", rate impact: which are severe?
- Suggestions, not mandates
- Share learning:
  - Provide references when introducing new concepts
  - Praise a new or interesting idea
- Take advantage of review tools like GitHub:
  - "Suggest change" features make it easy to incorporate small revisions

Intermission: Language exercise

- Why would you use THAT function?!
  - Try: Should this be other_function instead?
- This is incredibly sloppy work
  - Try: It would be helpful if there were tests. Please consider cleaning up the unused code before submitting, as this will make it easier to review and maintain.
- Is this supposed to do anything other than make me cry?
  - Try: I am having trouble following this function. Can it be simplified? If not, a docstring may be helpful for future reference.

Guidelines:

- Discuss the code, not the person
- Phrase as a question: leave room for different (good) alternatives
- Be clear about what is suggested
- Show the intended benefit
- Ask questions to learn, not to attack

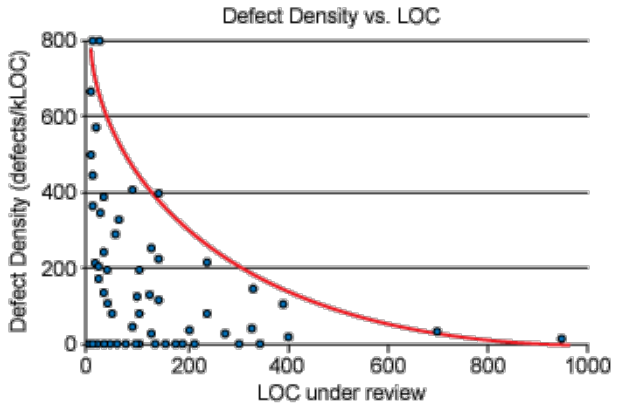
Finalizing a review

- Triage your requests:
  - Mandatory fix vs nice to have
  - What is achievable with current resources?
- Summarize comments: make concise and readable
- Define an action plan: Be willing to negotiate timelines and responsibility
- Schedule follow up if necessary

The role of the code author

Think incrementally

- Design or strategy questions can be discussed **before** formal review
- Clean up code as you go- continuous self improvement
- Don't overwhelm the reviewer: split code and reviews into small pieces; reviews are routine instead of rare

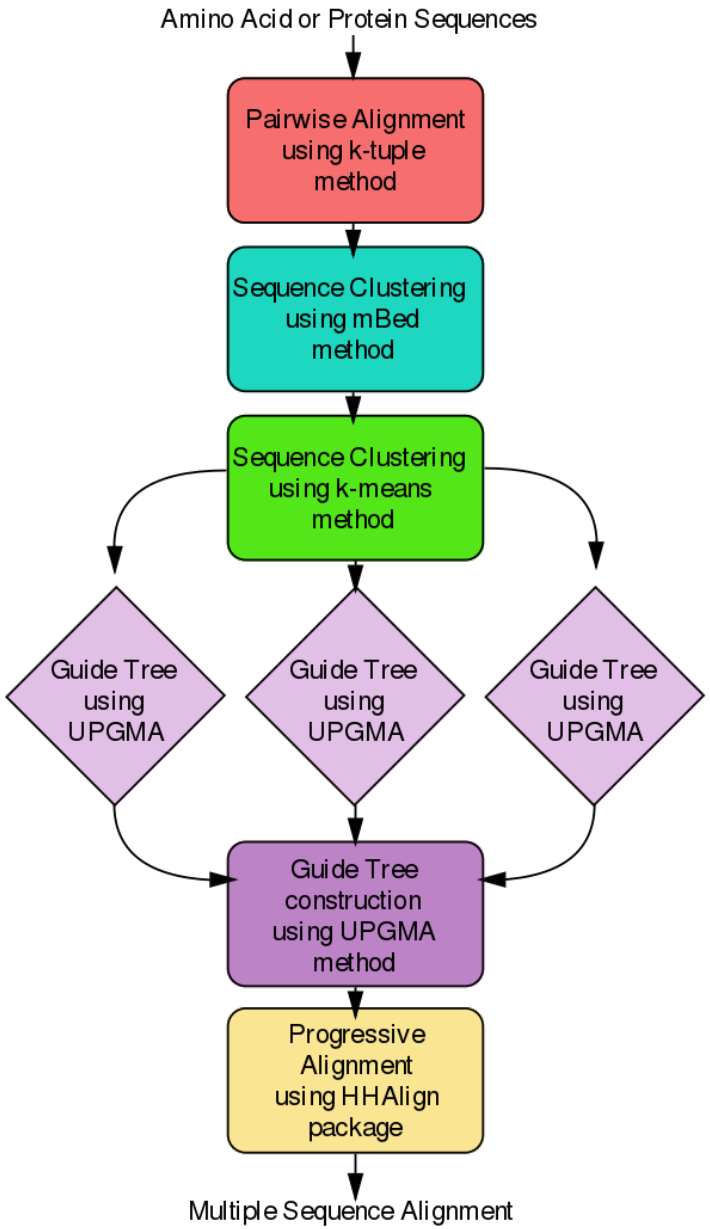


"...reviewing faster than 400 LOC per hour results in a severe drop-off in effectiveness."

https://www.ibm.com/developerworks/rational/library/11-proven-practices-for-peer-review/index.html

Preparing code to be reviewed

- Verify that code runs and look at UI. (don't **assume** it will work)
- Track required changes (can/should they be automated?)
  - Database migrations, dependency upgrades, system configuration, etc
- Write testing notes: help the reviewer identify areas where you want feedback
  - Create automated validation (unit tests) if possible
- Capture required artifacts- provide small test data when "archive" / "private" folders are not accessible



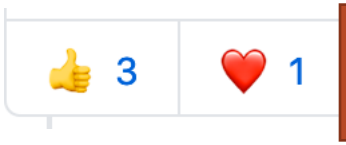
There is more than one way to convey your algorithm and intent to the reviewer.

More information: https://www.lucidchart.com/pages/uml-activity-diagram

Credit: https://commons.wikimedia.org/wiki/File:Clustal_Omega_Algorithm_Flowchart.svg

Responding to review

- Code review comments are a dialogue
  - You don't need to say yes to everything; justify your choices
- Be courteous and thorough. Did you cover all major points?
- Be prepared to discuss specifics if necessary
- Triage: can these changes be made separately later?
  - Follow through on your promises
  - Don't leave the problem for your reviewer to fix



TIP: GitHub emoji can act like checkmarks on individual points- "I saw this", "I responded to this one"

The role of the Team

Automate the boring stuff

- Human reviewers are expensive and valuable
  - Focus their time on what matters!
- Use agreed-upon tools and settings to handle routine tasks
  - Code formatting tools for style
  - Static analysis (linters) to catch common problems
  - Other quality checks (CodeClimate etc)
  - Continuous integration to ensure all checks are run

ESLint

axe
THE ACCESSIBILITY ENGINE

- Flake8/pylint
- mypy
- bandit

lintR

clang-tidy

GitHub Actions

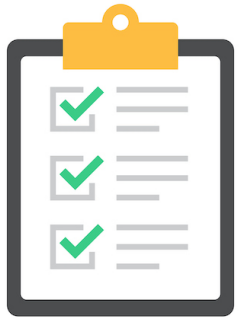
Identify barriers to automation

- Bad habits make it hard to use good tools
  - If you can't install/run code, CI won't work
  - Avoid: hard-coded paths or undocumented configuration
  - Database setup/migrations should be automated so that every test is run against a realistic environment
  - Code should be broken into pieces that allow unit testing
- Make small improvements in each cycle
- Maintenance required: Automation is a shared responsibility

Create shared expectations

- Good guidelines are:
  - Realistic and based on the needs of the project
  - Agreed upon by the team, rather than imposed
  - Subject to revision over time
- Set a culture of communication
  - Respectful: criticize code, not people
  - Flexible: open to discussion and alternatives
  - Curious: ask questions for the sake of learning

Example checklist: Web Applications



- **Features**: are complete and work as intended
- **Documentation:** code comments, docstrings, and specs
- **Compliance:** meets regulatory obligations (IRB, accessibility, etc)

- **Maintainability:** change is readable & thorough; include artifacts & tests
- **Performance/scalability**: site won't crash under heavy use
- **Privacy/security**: access controls, code injection, etc
- **Risk assessment**: can a change be rolled back if it causes problems?

Not listed: Architecture. Discuss that *before* you start coding.

Different checklists for different audiences

- General code review:
  - https://www.michaelagreiler.com/code-review-checklist-2/
- By and for scientists:
  - http://fperez.org/py4science/code_reviews.html
  - https://uwescience.github.io/neuroinformatics/2017/10/08/code-review.html
- Journal of Open Source Software: manuscripts about code
  - https://joss.readthedocs.io/en/latest/review_checklist.html
- OWASP Security Guides
  - https://owasp.org/www-project-top-ten/
  - https://owasp.org/www-pdf-archive/OWASP_Code_Review_Guide_v2.pdf
- Accessibility
  - https://www.a11yproject.com/checklist/
- Language specific style guides
  - https://google.github.io/styleguide/cppguide.html

Making it work

Common challenges and how to address them

Challenge: Process/alignment

People may come in with different expectations and can't agree on whether an item needs to be addressed.

Possible causes:

- The feedback is burdensome or does not address a specific problem
  - Eg automated tools that cause too much friction: fix or remove them
- Project planning issues
  - Delivery deadlines must include time for review and testing
  - Poor sense of long-term goals for project ("sure this function uses a lot of RAM, but it works on my sample file!")
- Review is seen as a "final grade" instead of an ongoing process
  - The team should create a set of shared guidelines, and revise them over time

Challenge: Communication

Review is a dialogue. Sometimes additional information is required to turn notes into action: the problem/fix may not be immediately obvious. Follow-up is required.

For the reviewer:

- When introducing new concepts, can you link to a good explanation?
- Is the fix within the technical abilities of the developer?

For the code author:

- Keep good notes so you can explain what was already tried, and what didn't work
- Don't try to bluff the reviewer or "make the problem go away"

For everyone:

- Be willing to say "I don't know", **and then learn**

Challenge: Advocacy

Sometimes, a problem is important, and even urgent to fix... but the solution is tedious. Someone needs to advocate for the work.

- Identify the constraint: time? Resources? Specialized expertise?
- Create a plan for implementing the fix, and **expect to follow through**
  - Designate a specific person who is accountable, and set a time to report back
- Avoid "maintenance" as a separate job: elevate continuous improvement

Challenge: Domain

The code may be hard to review due to required domain knowledge or highly specific optimization. Pieces will be complex and interconnected.

- "Correct" output may be unknown: capture validation as you go
- Algorithm may be hard to determine solely from code: how to document?
- Different baseline testing approach: common SWE practices must adapt
  - Author was focused on testing the science, not the code
  - Code may only be subjected to review at the end of the process
  - May have only been validated on huge slow datasets: how to iterate?

Jay, C. *et al.* The challenges of theory-software translation. *F1000Res* **9**, 1192 (2020). (link)

Good Code Review is:

- A way to share ideas and experience
- A peer activity focused on outcomes
- Small, incremental, and frequent
- A shared responsibility for everyone
- Most effective when used with other good habits

Further reading: Code Review

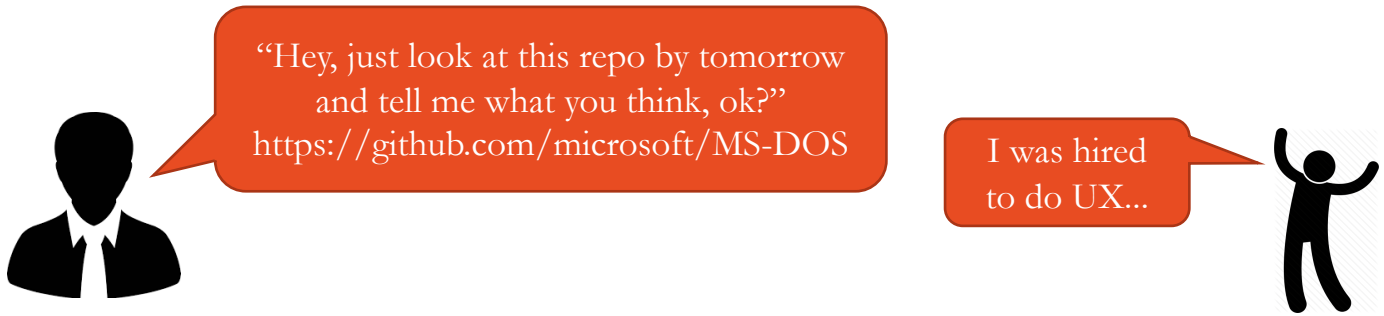
- How to read other people's code
  - https://selftaughtcoders.com/how-to-quickly-and-effectively-read-other-peoples-code/
- Conducting effective code reviews: Tips, phases, and sample ideas
  - https://auth0.com/blog/conducting-effective-code-reviews/#Different-Aspects-of-a-Code-Review
- "Software Engineering at Google: Ch 9"
  - https://learning.oreilly.com/library/view/software-engineering-at/9781492082781/ch09.html#code_review-id00002
- How to make good code reviews better
  - https://stackoverflow.blog/2019/09/30/how-to-make-good-code-reviews-better/
- MIT student code review guide
  - https://web.mit.edu/6.005/www/fa15/general/code-review.html

Further reading: Software Engineering

- Good enough practices in scientific computing
  - https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1005510
- US Research Software Engineer Association
  - https://us-rse.org/newsletters/
- Software Sustainability Institute (Guides on many topics)
  - https://www.software.ac.uk/resources/guides/guides-developers
  - https://software.ac.uk/sites/default/files/SSI-SoftwareEvaluationCriteria.pdf
- Prior presentations
  - Unit testing: https://github.com/abought/slides/blob/gh-pages/csg/automated-testing/boughton-software_testing_slides_2019_07_30.pdf
  - Debugging: https://github.com/abought/slides/blob/gh-pages/csg/debugging-techniques/boughton_debugging_techniques-2020_02_25.pdf

When review fails

- Unclear expectations: poorly described code, no shared goals
- Diffuse teams: No clear responsibility for fixes or follow-up
- Barriers to entry: eg specialized infrastructure/ secret data that takes longer to set up than the actual code review
- Undue burden: eg too much code all at once
- Knowledge transfer: difficult on small teams, or with long time lag between authors



"Just another voice in support of the notion that code review should be focused much more on knowledge transfer and shared good practice (i.e. the learning is bidirectional) than on bug finding. In an environment where **most code is written by students, who have a known rate of turnover**, it's a way to guarantee that successor students have an understanding of the code that will become their responsibility and the basis of their research projects. There's a corollary to this, which is that **students can't be allowed to carry their code too far "out into the wilderness"**... Otherwise, you'll have students who spend the last year or whatever before they graduate grinding out the last... results... and then they leave with no one else ever having looked at it..."

-Phil Miller, via USRSE Slack

“Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”

-Kernighan's law