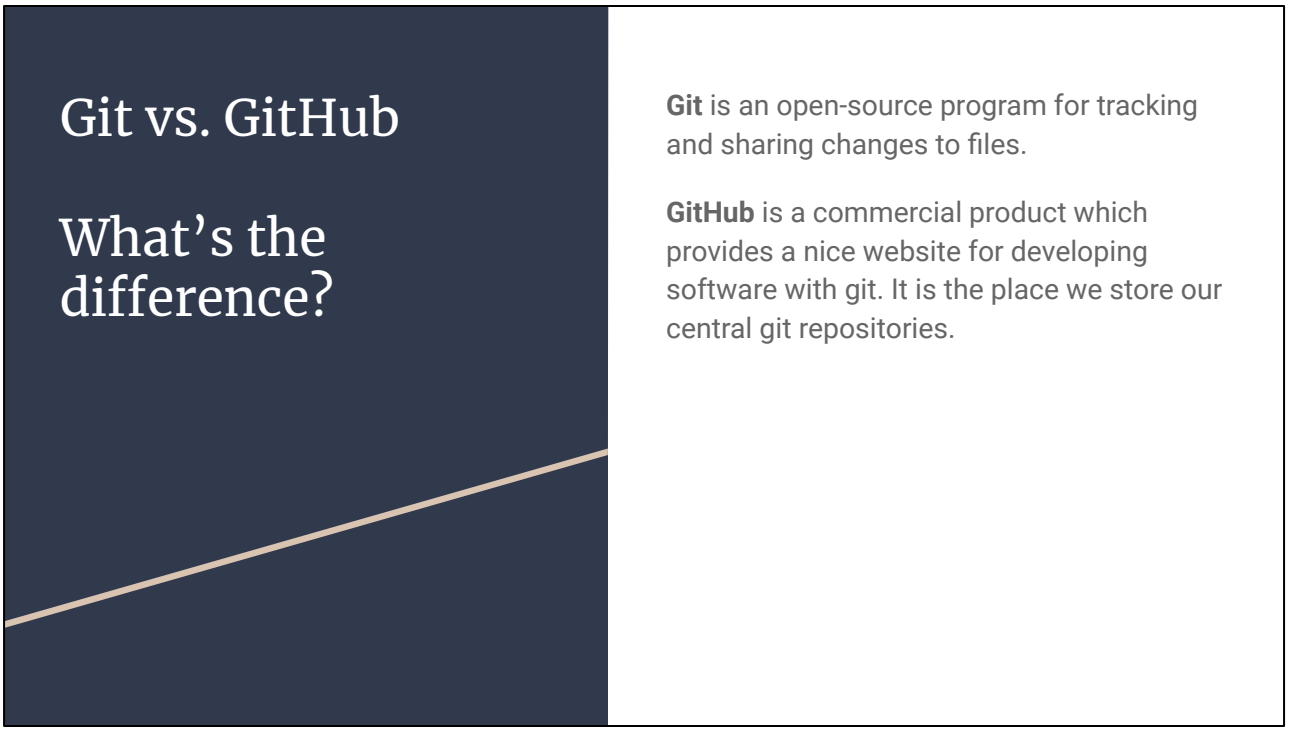
Making the most of version control

Git best practices and development hygiene

Thomas Sibley
Bedford Lab
3 August 2018

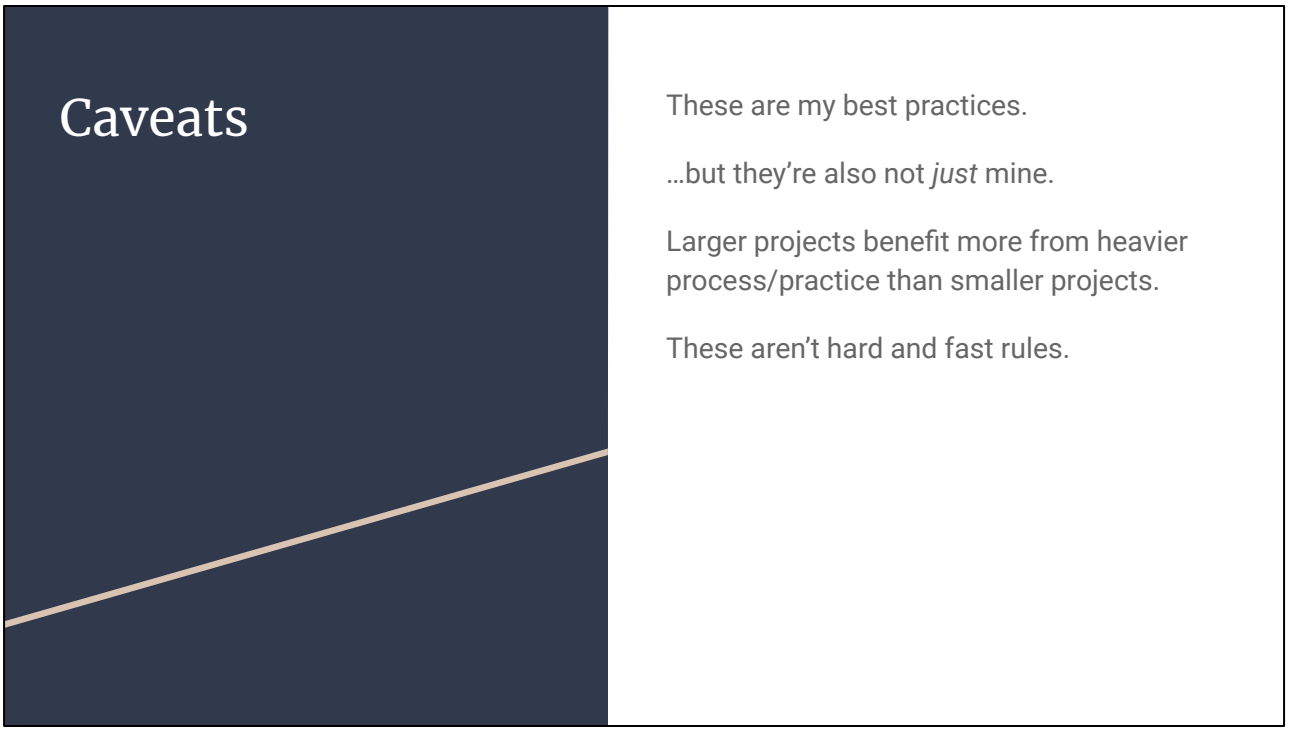


I want to lead off with some quick background which might be old hat to some but maybe isn't crystal clear to all, which is the difference between git and GitHub.

When you're using the command-line to check your repo status, view the changes you've made so far, commit your work, or pull down updates, you're using git.

When you use the web browser to comment on a PR or browse some code, you're using GitHub. GitHub is where our central repositories live and where other people find our code.

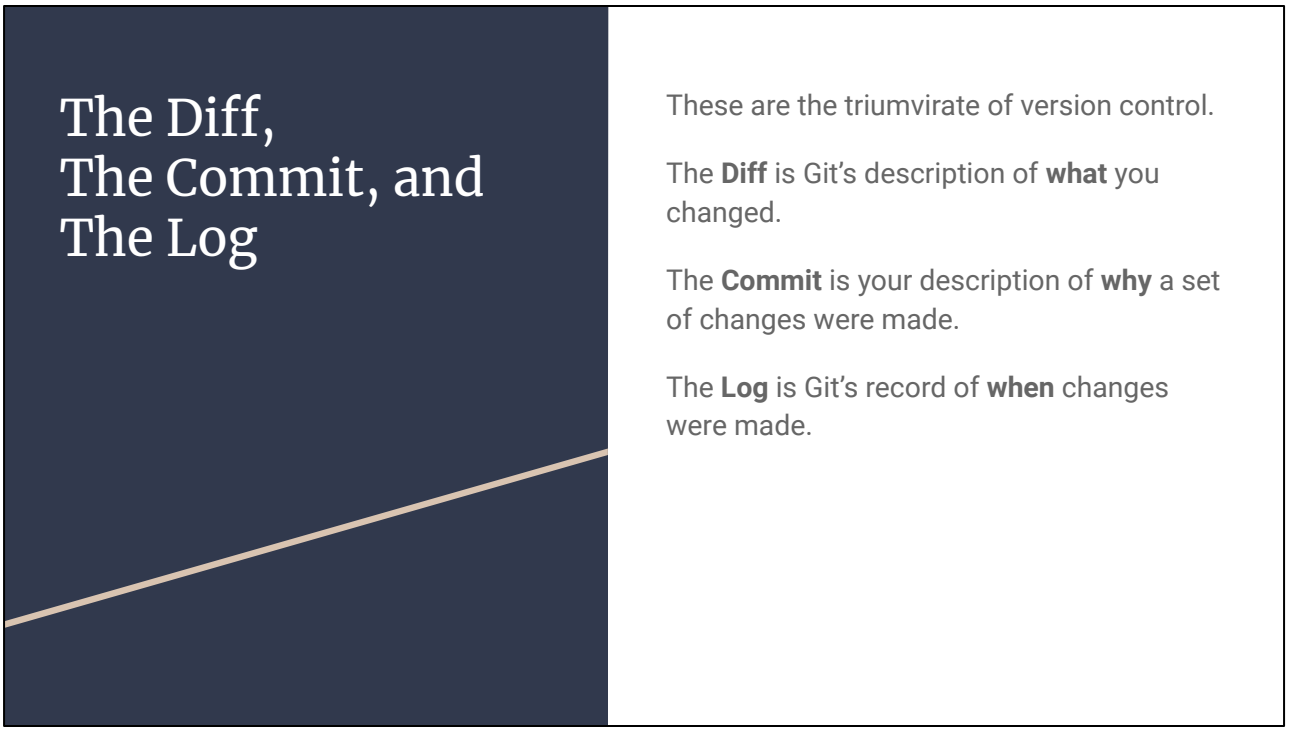
I'm mostly going to be talking about practices that apply generally to git, not just GitHub. And while I'm focusing on git, the same principles apply to other version control systems too.



I also want to lead with some caveats about what follows. The practices I'll be talking about are definitely based on my own experience with what works well, but I'm certainly not the first to articulate them. There is a long history of thought about how to improve the (often painful) process of writing software, and the practices I'll be talking about have fairly wide acceptance.

What I'll be talking about especially applies to larger software projects with lots of moving parts, like the Nextstrain ecosystem. While these practices apply to smaller scripts and notebooks as well, the cost-benefit analysis is different and not as clear cut.

All of which is to say that what's to come aren't hard and fast rules. They're guidelines and ways of thinking about version control that offer advantages to future development and make it easier for outsiders to get started contributing to your project. Use them in projects where it makes sense.



(Describe each.)

I'm going to touch on each of these in turn and best practices for cultivating them. Each builds upon the previous ones.

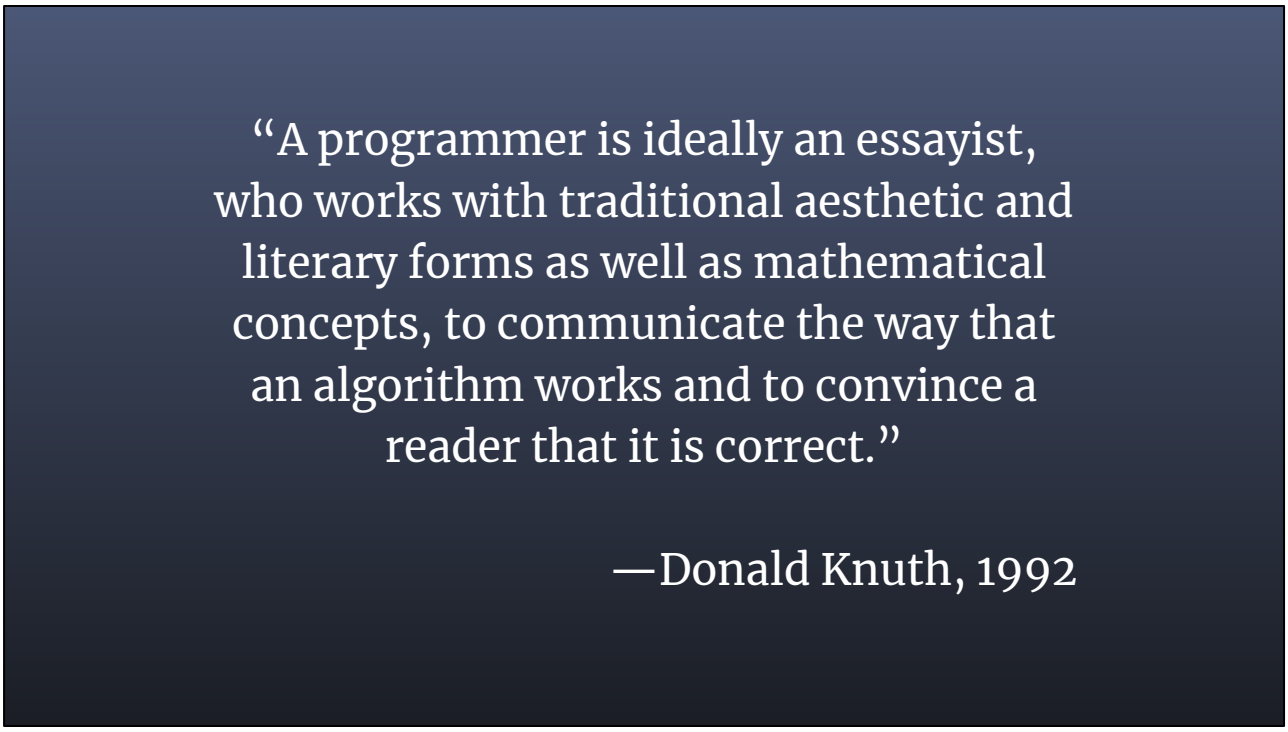
The Diff

Programming is Writing

Draft, revise, cut, rinse, repeat

Don't stop at the first thing that seems to work
- It may not work in all cases!
- It is probably not the clearest way to communicate the code's intent.

Read and re-read your diffs frequently
- You'll notice improvements
- Hone them for clarity and correctness



I came across this quote from Donald Knuth recently and quite liked it.

The Commit

Why not *What*

Summarize **what** changed briefly, and then describe in detail **why** it changed and **why** you made the choices you did.

The Diff describes in detail **what** changed, but only you can describe **why**!

Good commit messages are a love letter to the future.

Avoid `git commit -m`. Configure git to use your text editor instead.

(Why good commit messages are so important.)

Empowering the log.

The future will often be you, 6 months later, trying to figure out why you did something 6 months ago.

```
Summarize changes in around 70-80 characters

More detailed explanatory text, if necessary.
Explain the problem that this commit is solving.
Focus on why you are making this change. Are there
side effects or other unintuitive consequences of
this change? Here's the place to explain them.

  - Bullet points are okay, too…

  - …if they’re not just enumerating unrelated
    changes

Links to further discussion that informed this
commit can be useful. If you use an issue tracker,
put references to them at the bottom, like this:

Resolves: #123
See also: #456, #789
```

- “Also, …”
- “While I was at it…”
- Whole message is a bulleted list

The Commit

Split up your changes

Don’t lump changes together simply because they happened close in time.

Commits should be discrete, cohesive chunks with a clear purpose.

Make commits incrementally, even from work done simultaneously.

```
git add --patch
```

```
git commit --patch
```

```
git rebase --interactive
```

The Log

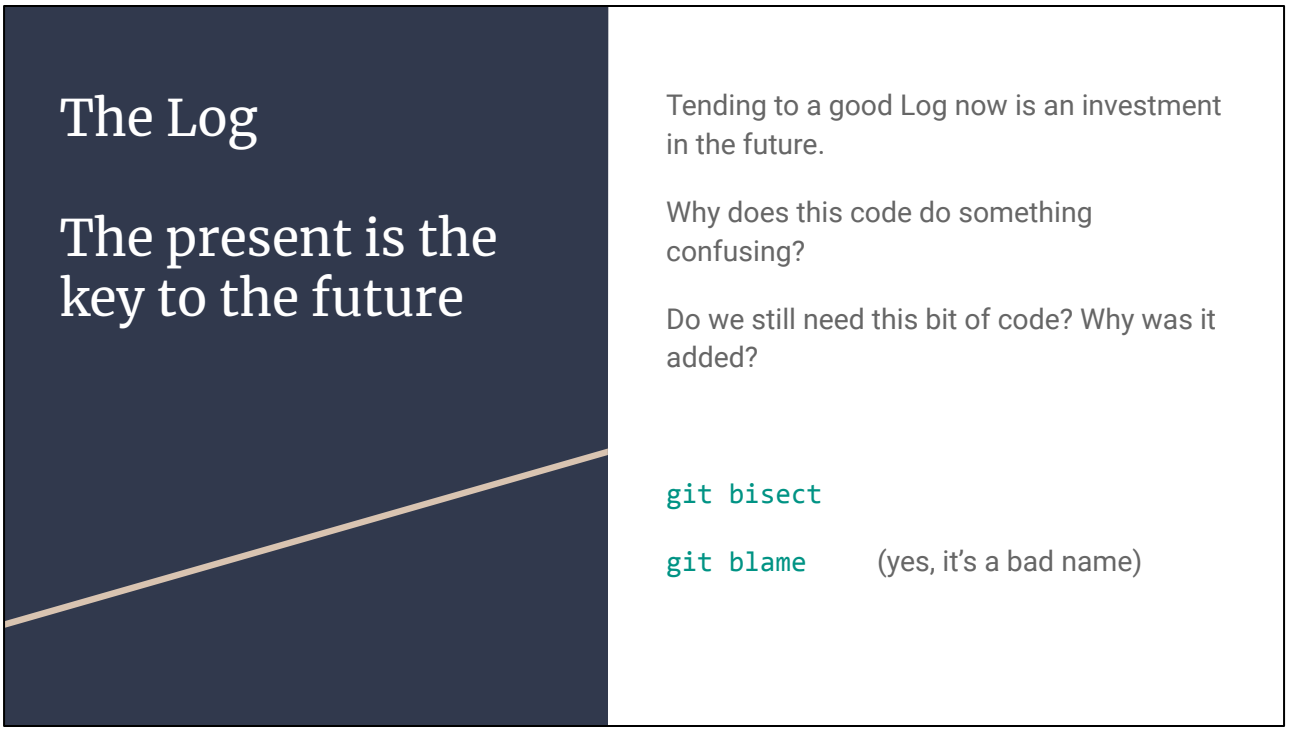
The past is the key to the present

xkcd.com/1296/

Charles Lyell, Scottish Enlightenment (18th-19th century)

Importance of history

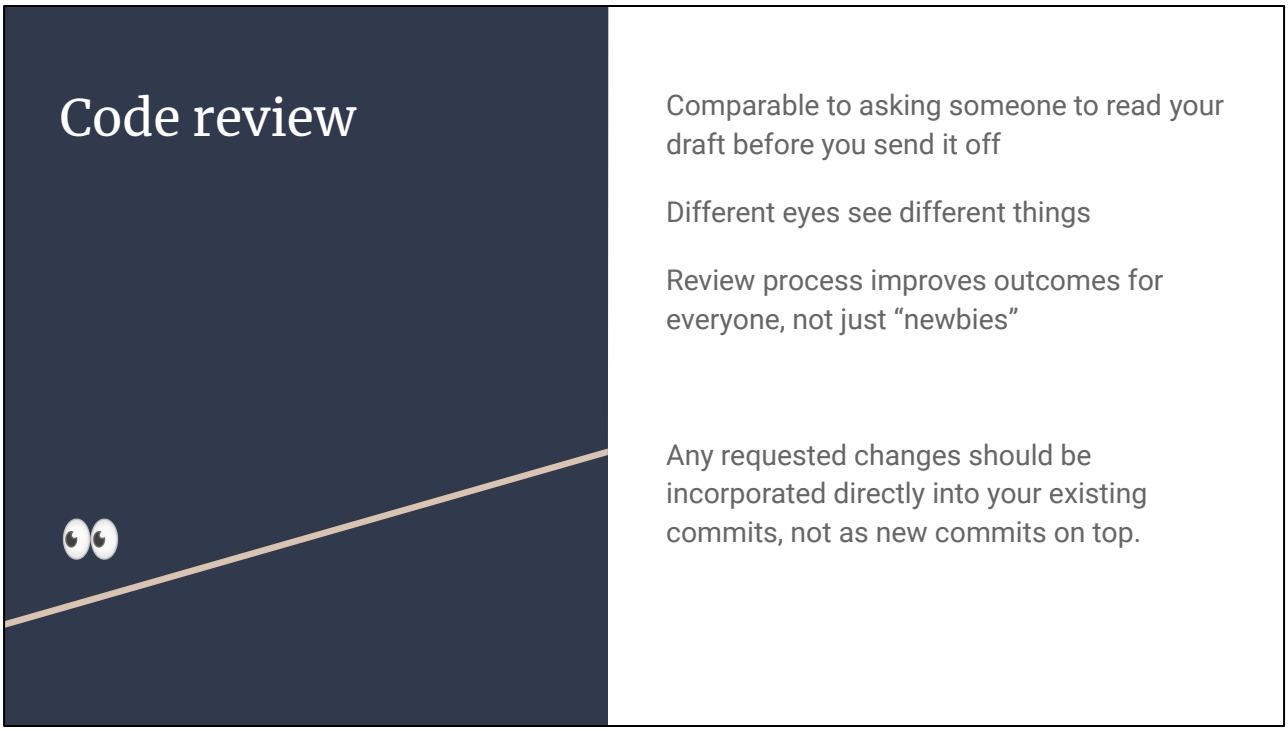
This is a rather extreme example of a poorly tended git log from XKCD to get a laugh, but poorly tended logs don't have a clear sense of direction. A good log should tell a story. This is different from your train of thought while hacking on code, where there are dead ends and distractions along the way. Using the commands from the previous slide you can craft that story after the fact, once you've done the work and know how the story should end.



Writing good commits now and tending to a clean log is an investment in the future and the only way to get to a useful history.

There are many git commands which rely on a useful history, such as bisect for automatically finding when a bug or behaviour started happening or blame/annotate for tracing the origin of a bit of code.

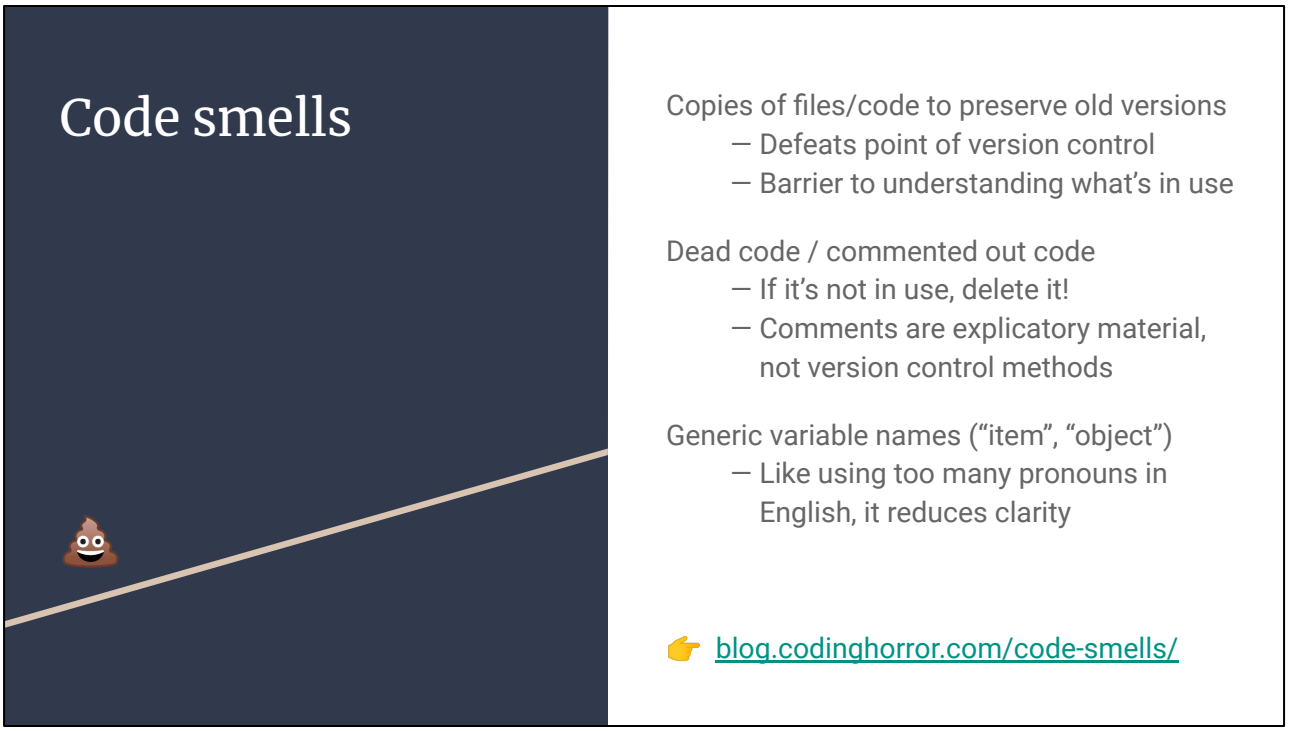
The messier a log is, the less useful it is, and so there's less reason to tend to it. But investing in a good log will provide a valuable tool for longevity in a software project.



One way to improve on all of these things I’ve talked about is via code review. This comes back to the Programming is Writing philosophy. Code review is just like asking someone to read your draft of a manuscript. Can they follow the story you’re trying to tell? Do they understand why things happened? Importantly, just like editing a manuscript, the code review process is for everyone; it’s beneficial to experts as well as newbies.

(Material on slide)

When working on a document, you don’t tack on edits at the end, you incorporate them into the body of work. Your code should be no different.



One thing that comes up in review can be code smells. Code smells are things that are indicative of poor development practices. They often don't affect current functionality, but they hinder development and future changes, often by decreasing clarity or flexibility or increasing the frequency of bugs.

XXX TODO

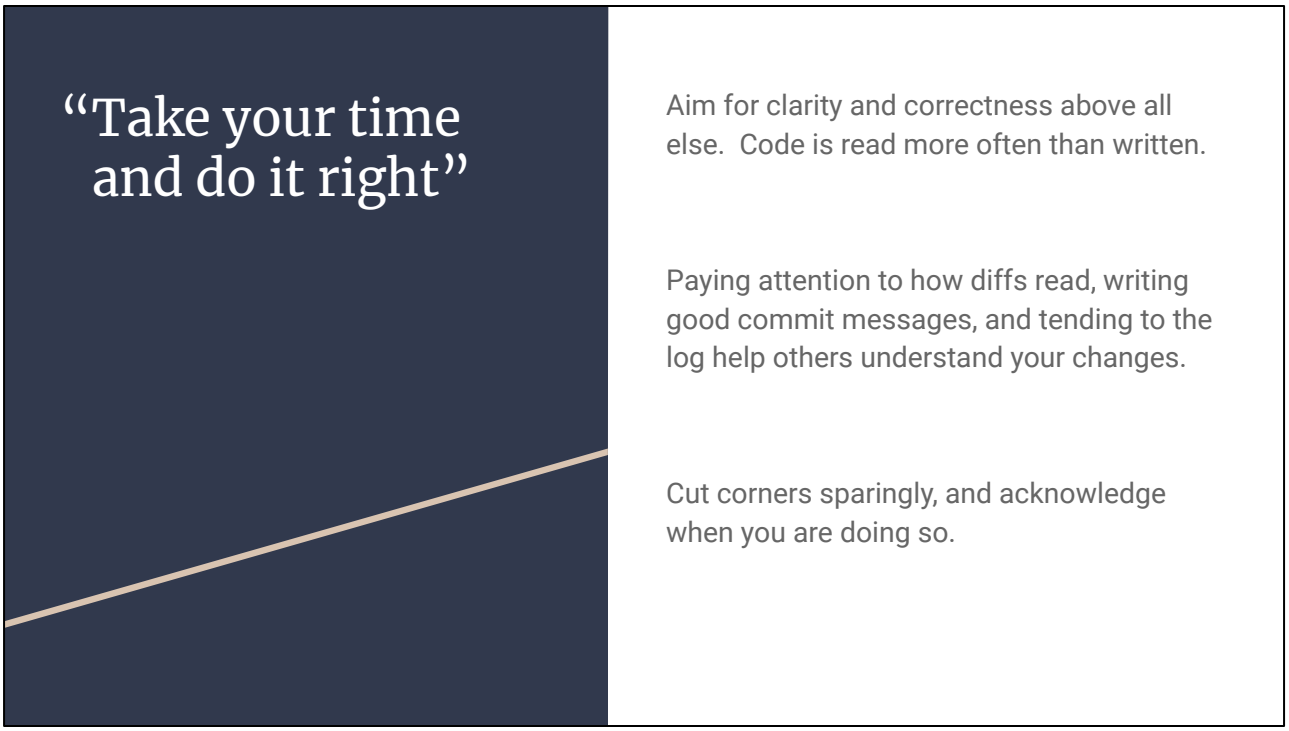
The link has a reasonable list of many common code smells, but know that code smells are very subjective and variable project-to-project. For example, in a combined code plus data repo for a manuscript, it may be completely justified to use multiple copies of files as a way to have side-by-side versions of a dataset or resultset.

“Move fast and break things”

“We used to have this famous mantra. [The idea] is that as developers, moving quickly is so important that we were even willing to tolerate a few bugs in order to do it. What we realized over time is that it wasn’t helping us to move faster because we had to slow down to fix these bugs and it wasn’t improving our speed.”

—Mark Zuckerberg, 2014

mashable.com/2014/04/30/facebooks-new-mantra-move-fast-with-stability



If there’s a mantra I’m more fond of, it might be “Take your time and do it right.”

Clarity and correctness are hard to come by, but they’re harder to beat. There’s a time to cut corners in every project, but do it sparingly and only when necessary to meet a real deadline.

This leads into my final point, which is…

Leave the code better than you found it.

Leave the code better than you found it. Not just in what new things you add, but also in fixing things along the way that you notice are broken or could be done better.


"That's all Folks!"

Rebasing:

What changed?

git tbdiff exists to show you what changed between two versions of a topic branch.

git range-diff is the new, core version of this!

Topic branch is a fancy term for a branch that adds a feature or fixes a bug. It draws a distinction between those short-lived branches and branches that persist indefinitely, like “master”.

👉 github.com/tsibley/git-tbdiff

Further reading

chris.beams.io/posts/git-commit/

git-scm.com/book/en/v2/Git-Tools-Rewriting-History

git-scm.com/book/en/v2/Git-Tools-Reset-Demystified

sourcemaking.com/refactoring/smells

sourcemaking.com/refactoring/refactorings