# Avoiding Groundhog Day

*A practitioner's daily mentoring through code reviews*

Michael Tautschnig

# Caveats

1. This is my area of practice, not research.
2. My academic background is formal verification.

# What I thought:

*Code review* is a quality-assurance step.
(And I thought: it's a poor quality-assurance step.)

# Code review

From Wikipedia, the free encyclopedia

This article **is written like a personal reflection or opinion essay** that states a Wikipedia editor's personal feelings about a topic. Please help improve it by rewriting it in an encyclopedic style.
*(October 2017) (Learn how and when to remove this template message)*

**Code review** is systematic examination (sometimes referred to as peer review) of computer source code. It is intended to find mistakes overlooked in software development, improving the overall quality of software. Reviews are done in various forms such as pair programming, informal walkthroughs, and formal inspections.[1]

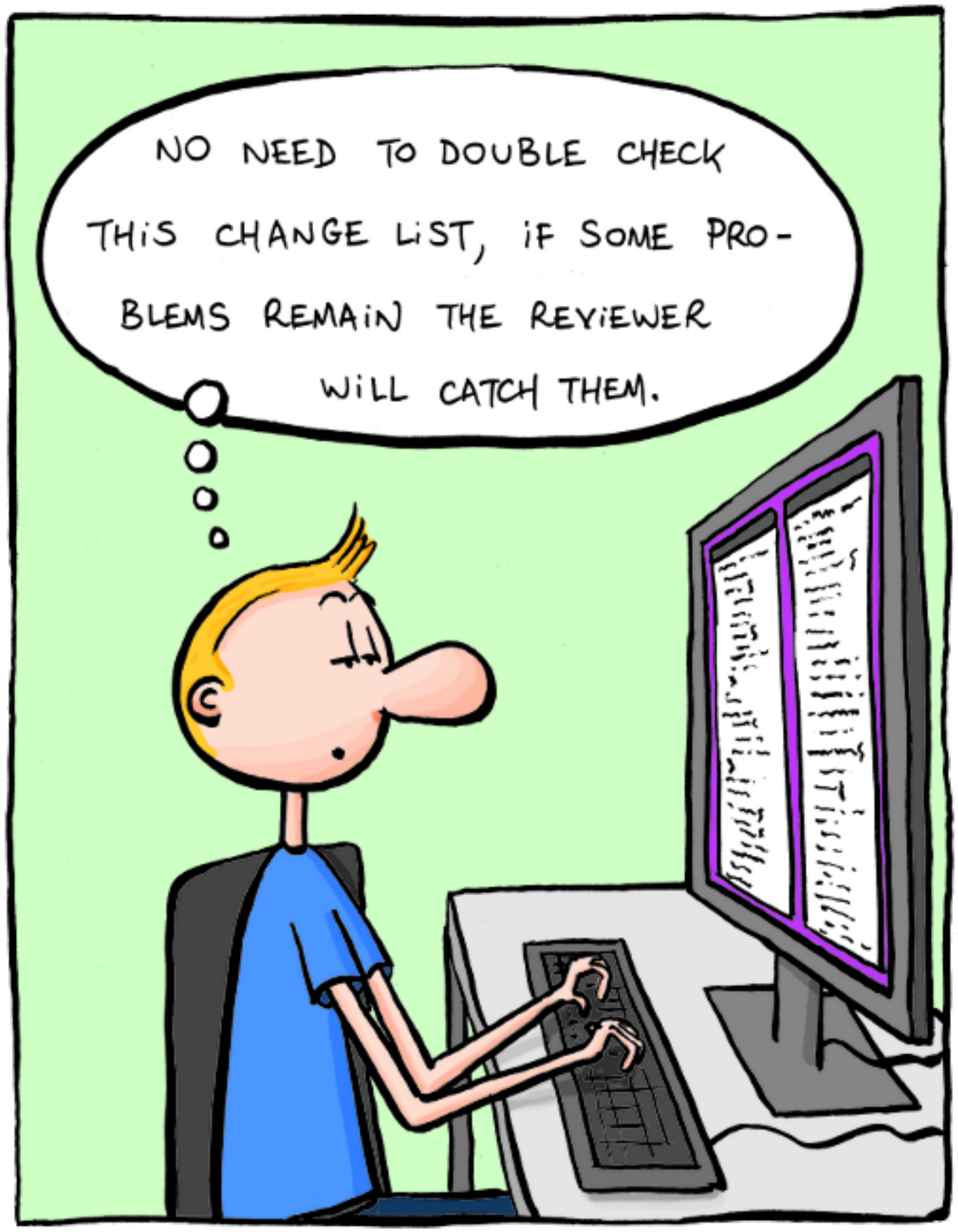
https://en.wikipedia.org/wiki/Code_review

# Over the last 28 months I have ...

- Submitted > 260 + 324 pull requests for review.
- Received > 1600 + 1090 pull requests for review.
- Commented on 500? + 549 pull requests.
- Learnt that code review is 1) more useful than I thought and 2) useful in a very different way.

# What I learned:

Code review is a communication tool.

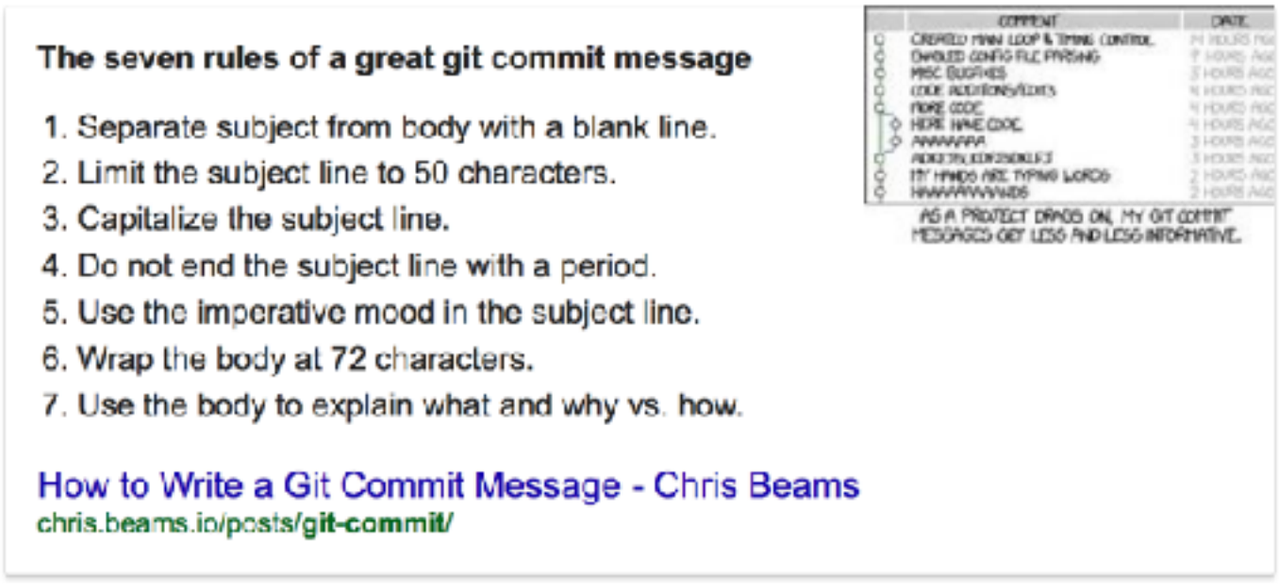




https://medium.com/@Andela/whats-the-real-essence-of-code-review-a5867109a1e1

# Why Pull Requests?

- Synchronisation point
  - *What* changes?
  - *Why* does it change?
  - *How* does the code change?
- Enables review with *limited context*

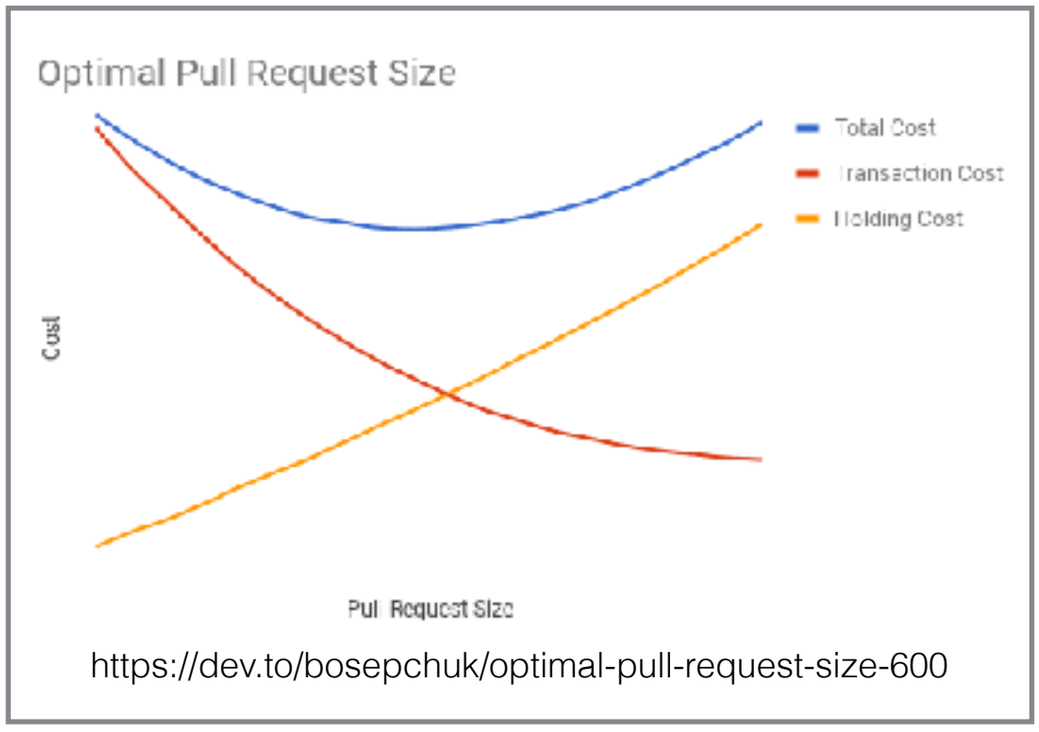


# The ideal PR+CR Process

- React quickly, both with reviews and replies / refinements: Preserve momentum, minimise human context switches
- Involve stake-holders and domain experts, spread load
- Ping people if necessary
- Help reviewers decide whether your change is good
    - Clean code
    - Good description
    - Demonstrate soundness (simple is better, testing)
- Be respectful of reviewers' time → make it easy

# Good PR Patterns

- Small, incremental changes: One PR per feature
- Break up long PRs
    - Clean-ups (whitespace, typos, coding style)
    - Refactoring changes
    - Functional changes
    - (Almost) no too small CR
- Preserve bisectability: each commit should compile and run
- Sensible testing → context-free judgment of correctness
- Early idea → Mark PR as RFC
- Good commit messages



https://dev.to/bosepchuk/optimal-pull-request-size-600

# Good Commit Messages

- Good description
  http://chris.beams.io/posts/git-commit/

- Meant for eternity, read-mostly (optimise for that)

- Expression, grammar, spelling, capitalisation

- *Why & what in message, how in diff*

- Linux-style git workflow, chain-of-custody tags
  https://www.kernel.org/doc/Documentation/SubmittingPatches

# Why code review?

https://medium.com/@Andela/whats-the-real-essence-of-code-review-a5867109a1e1

- To facilitate *knowledge sharing* across the code base and across the team.

- To *ensure maintainability* of code—hence the project, especially when a team member has to leave the project.

- To *learn new technologies and techniques*, as we're never always on the same level of knowledge with all of our teammates.

- To reduce *code smells*.

# … and to avoid Groundhog Day

# Code review is educating

**tautschnig** requested changes 18 days ago • edited

View changes

Let me try to be constructive:

1. https://dev.to/bosepchuk/optimal-pull-request-size-600
2. The linter speaks for itself.
3. https://chris.beams.io/posts/git-commit/ (I dislike repeating myself, FWIW).
4. ~~Introducing new APIs that do not include comments (see 3 concerning the corresponding commit message, which could have been a fallback option) is a no-go.~~

While I do get the overarching topic of this PR, for the reasons laid out in 1 you should not mix up different bits of work. Specifically, this PR has at least 3 steps: a) the use of `unique_ptr`; b) moving from enums to distinct types; c) the use of a visitor. I'm not saying that any of this is wrong, but together this resulted in 32 commits that represent how this historically evolved rather than separate topics and ~~+2,726 −1,628~~+2,779 −1,628 changes.

2 and **4** above are blockers. I'll dismiss this blocker once those are addressed and then maybe someone else is willing to thoroughly review this.

~~Causing an ICE with Visual Studio is also a blocker, of interesting sorts...~~

Addendum: I get the idea of type safety, but I'm a bit surprised that this required more than a 1000 extra lines of code. Nothing blocking, but still curious.


http://softwareandotherthings.blogspot.ie/2015/08/make-your-code-reviews-agile.html

# Code review is architecting

src/goto-symex/goto_symex.h (Diff revision 1)

49 symex_target_equationt equation);

this will be very expensive, I believe

[redacted] 1 month, 1 week ago (Oct. 17, 2017, 1:00 p.m.)

Thanks for these comments! I'll address them all in spare moments, but about the cost of saving states---what do you think about serialising states to disk?

Both members of this branch pointt thing are mightily expensive, and it's more and more seeming that I could spend forever making micro-optimisations that will not address the fact that this still won't be scalable on bigger programs. Even if I find some members of goto symex statet that aren't needed for path exploration, I do need most of the members of that object, I don't think it's big without reason. And I think saving the entire equation is the right choice, somehow. The two alternatives to keeping the list of all SSA steps that took us to this branch point so far are:

- save only a list of branch decisions, i.e. a vector of booleans, and re-execute the program up to the save-point when we wish to take that path. Very space-efficient but seems like a huge waste of time.
- implement 'diffs' of SSA steps, i.e. something like having a pointer to another symex_target_equationt so that we never duplicate subsets of paths. However, this isn't actually going to save any memory unless we occasionally delete subtrees that we've finished exploring. As Mark pointed out, that means that I would need to implement garbage collection, which isn't terribly productive.

I could sidestep all of these issues by dumping both the symex_target_equationt and goto_symex_statet to disk and deserialising them when needed. I wouldn't have to think about space efficiency, and the code would hopefully be simpler and more maintainable than doing clever tricks with re-execution or sharing SSA lists or whatever. Even if serialising needs to be complicated, it's somewhat orthogonal to the rest of the implementation and wouldn't clutter the rest of it up; consequally, we're still just popping a state off the workqueue and resuming, except that behind the scenes we'd be swapping disc to do that.

Thoughts? My only concern is that to serialise those two classes, I'd also need to implement serialisation for the classes of each of their members, and so on transitively. Not sure how many classes that is, and what weird issues I'd need to deal with (e.g. std::unique_ptr and the like).

Michael Tautschnig 1 month, 1 week ago (Oct. 15, 2017, 1:34 a.m.)

I think serialising to disk is an option when 1) you rarely need to re-load them, as obviously this might be rather slow, and 2) memory indeed becomes a key limiting factor (as opposed to the mere CPU cost of having to explore so many branches). Hence I'd challenge you to add some measurements to your plan in order to make data-driven decisions here.

On the technical side of implementing serialisation: you're all set for irept's, you just need to figure out how much extra data you need to store to be able to reconstruct the structures around them (e.g., the list that a symex_target_equationt is). The format that is used for goto binaries is actually pretty simple, and I'm sure you could re-use a lot of that.

As a side note: if you figure out how to serialise what you just described, then you're half way done for snapshotting and re-starting symbolic execution (irrespective of whether you work in a single-path or path-merging set up), which would of course also be very welcome, as discussed in the past :)

[redacted] 1 month, 1 week ago (Oct. 16, 2017, 6:49 a.m.)

Thanks, that makes sense, so I believe my next plan of action should be to get this (expensive) implementation working ASAP; benchmark it on SV-COMP and other things; and figure out exactly where the pain points are. Does that sound good?

About reading from disk being slow: my idea was to load and save states asynchronously, while CBMC is doing symbolic execution. We would have a data structure that looks like a worklist of states, but is actually a worklist of filenames of serialised states. The structure would have the "next" state to be popped already held in memory, and as soon as we pop that state the structure would create a new thread to load a new state from disk while CBMC is busy with the CPU. That way there's always one (or maybe a small number) of states held in memory, waiting to go, but most of them are on disk. Similarly, when adding a state to the worklist, the structure creates a thread to save the state so that CBMC can continue working.

Again, I'd want data to see how much of a choke-point disk access would be to justify implementing this, but since CBMC is single-threaded and CPU-bound this seems like it should be beneficial in cases where the program is big enough.

Michael Tautschnig 1 month, 1 week ago (Oct. 16, 2017, 7:17 a.m.)

Sounds good to me!

# Code review is learning

**src/goto-cc/gcc_mode.h** (Diff revision 2)

protected:

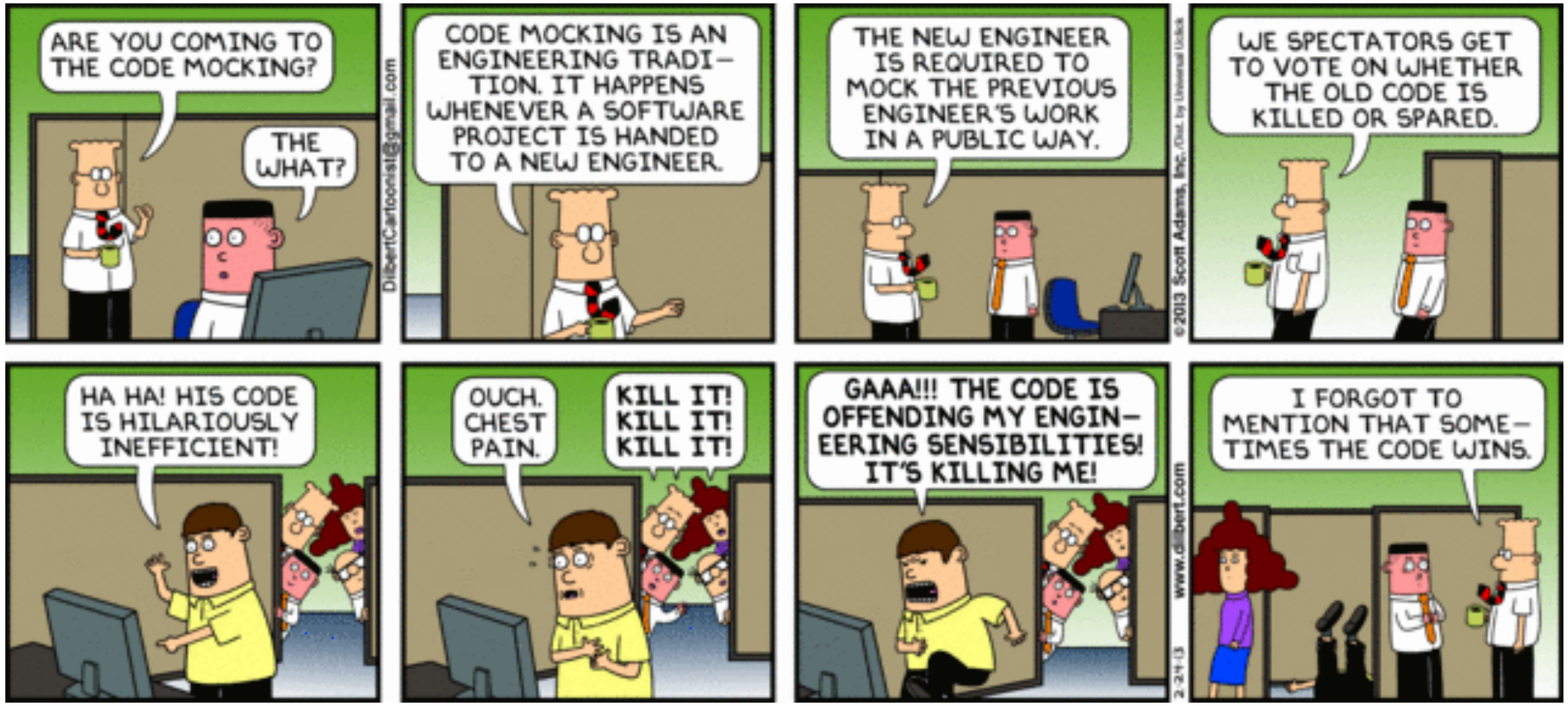
| | | | |
|---|---|---|---|
| 48 | `int run_gcc(); // call gcc with original command line` | 48 | `/// \brief call gcc with original command line` |

\brief should be used for one-liner comments, and for headers of longer doc comments. I think that's appropriate here, unless you think that this function shouldn't have a doc comment at all...in that case I'd just change it back to a regular (//) comment.

The reason I changed this comment at all is because I had to move it to the previous line (due to increased line length), so I thought I might as well change it into a doxygen comment...

**Michael Tautschnig** 4 months, 3 weeks ago (July 4, 2017, 8:45 a.m.)

Ah, cool, learnt one more thing then :-)
Keep it as you've done, sounds exactly right.




http://dilbert.com/strip/2013-02-24


- Code review is communication
  - Opportunity for learning, education, knowledge sharing
- I still think code review alone is an insufficient quality-assurance mechanism.
  - Could someone please fix the Wikipedia page?