Your Code Sucks and I Hate You: The Social Dynamics of Code Reviews

Jonathan Lange, September 15th 2008

Abstract

Code review can be wonderful, helpful and incredibly daunting. This paper looks at how you can do code reviews for your project without provoking your fight-or-flight instincts.

We quickly gloss over *why* you should do code reviews in order to focus on the social dynamics of *how* code gets reviewed, particularly in open source projects. After all, part of what makes open source so great (and sometimes intimidating!) is that your code gets eyeballed by experts around the globe.

We look at the effects that different technologies can have on code review culture, what reviews can actually achieve, how other disciplines do review and then highlight some traps that are all too easy to fall in to.

Introduction

Code reviews have been around for a long time and have been part of the Free Software movement since its beginning: at least in the form of those with commit privileges reviewing patches from those without.

In recent years, code review has gained new prominence. Extreme Programming has championed the idea of pair programming—a kind of continuous code review—and many open source projects now require all patches to be reviewed, no matter how trusted the author.

Each of these projects have different approaches to review: different tools, different processes and different aims that lead to different pressures on their development communities.

This paper explores the decisions these projects have made and the social pressures and incentives that result. It also identifies potential problems in review processes and how makes suggestions on how to avoid them. This paper is not intended to be a rigorous academic survey; rather it is a collection of considered opinions aimed to prompt further thought, research and experimentation.

What is a code review?

Simply put, a *code review* is what happens when someone looks at code and makes comments about it.

This paper focuses on *pre-merge reviews*, reviews that are performed on patches before they land on the main development branch. The classic Free Software example is a new developer submitting a patch to a mailing list. In some projects, the pre-merge review process continues even for experienced developers (e.g. Twisted, Bazaar, Linux).

There is such a thing as *post-merge reviews*. These happen a lot in Free Software projects after someone

has committed something dodgy to trunk. In general, the reviewer replies to the commit notification and highlights the error.

Reviews can also be done as a kind of *random inspection*. In this approach, a senior programmer inspects some part of the current code-base and makes comments about its structure and quality.

Some reviews are *involuntary*. These tend to happen late at night when a programmer is deciphering code in an obscure corner of the project, perhaps trying to fix a bug. These code reviews are often characterized by coarse language, angry comments and are generally unhelpful to the original author, who might not be fortunate enough to hear it.

What can reviews achieve?

For some people, code reviews fall into the same category as documentation, test-driven development and bran: good if dull things that we indulge in less often than we ought.

Unlike high-fibre cereal, code review can have many different and sometimes competing benefits. When doing code reviews in a project, it's important to consider which of these benefits actually matter.

Enforcing conformity

Code reviews are great for making sure all source code looks like all other source code. This includes naming conventions, correct spelling in comments and consistent use of internal APIs.

Ensuring quality

Reviewing code before it lands provides an opportunity to spot bugs before they start affecting users. A reviewer has more emotional distance from the code and is likely to look at it more critically, thus spotting more bugs.

Enhancing clarity

If a chunk of code has been reviewed by someone else, then there are at least two people in the world who can understand it.

A reviewer has more intellectual distance from the code and can spot hidden assumptions and unclear decisions.

A clear code base allows newcomers to start contributing quickly, which is particularly important for open source projects that rely entirely on volunteer effort.

Educating reviewers

If all code must be reviewed before it lands, then each patch offers a chance to learn about a part of the code base that they might not have seen before. This means that a review process can be harnessed to share knowledge about the code base across the community.

Having a totally fresh set of eyes review the code can be a mixed blessing: sometimes it will lead to

increased clarity; other times it will result in shallow reviews.

Balancing priorities

Since developers of a project often have different priorities, the person reviewing code will often have different priorities from the author. Someone might submit a patch that improves performance of a core API, but then have to justify why this is more important than preserving backwards compatibility. The code review forces this discussion to take place before the change becomes settled in trunk.

Making better programmers

A review by a programmer is an insight into how that programmer thinks, and prolonged exposure to the thinking of other programmers expands one's own thinking. Reviewers can suggest techniques that authors may not have heard of. They can ask questions that would never have been thought of. They can show an easier way that the author might not have seen.

On the other side, reviewing code also compels the reviewer to clarify what exactly makes good code.

Summary

The dynamics of code reviews will vary according to which of these goals is most important. If reviews are primarily about ensuring code *quality*, then reviews will take longer and be more thorough, as reviewers try to catch bugs, performance regressions and the like. If reviews are primarily about enhancing *clarity*, then reviews will be shorter, ask more questions, and tend to assume that the patch is well-tested and well-motivated.

Decisions

There are a few important questions that need to be answered when setting up a review process. Here we look at some of these questions and see how they are answered by the Bazaar, Launchpad and Twisted projects.

Who are the reviewers?

Perhaps the most important question that a project must answer is “Who are the reviewers?”. In most open source projects, reviewers are drawn from the pool of core contributors, although the exact mechanisms vary.

Patches to the Bazaar project need to be approved by two core developers, defined as people who can land these patches on trunk. This ensures that each patch is reviewed by someone with expertise, and also ensures a base level of communication between developers.

Launchpad has a reviewer team which forms a subset of the general Launchpad development team. The review team wants all developers to be reviewers, but has a process where developers must be “mentored” before becoming full-fledged reviewers. The mentoring process ensures that new reviewers have someone to turn to when they are unsure, and that Launchpad's review priorities (code clarity, fast turn-around of reviews) are held by all reviewers.

Twisted allows *anyone* to do a code review, as long as that person didn't have a hand in the patch itself. This rather clever idea prevents group-think between the author and the reviewer. The downside is that reviews tend to vary a lot between reviewers, and that because reviews are everybody's job, they can quickly become no-one's job.

Which forum?

Twisted does code reviews on bug tickets. Bazaar does code reviews on the general mailing list. Launchpad does code reviews on a separate reviewer-only mailing list.

Twisted's approach is an example of UQDS, the “Ultimate Quality Development System”. Reviews are done on tickets so that the ticket web page becomes the canonical authority for all information about a particular bug-fix or feature. As well as gathering information in one place, it can avoid bike-shed style discussions from people with no real stake. This benefit has an equal-and-opposite drawback: reviews themselves are rarely read by anyone other than the patch author, making it easier for different reviewers to apply different standards. If important questions are raised in a review, it is difficult to bring them to the attention of the wider community. Trac's unfortunate mailing system can make it hard to follow a discussion.

Bazaar has reviews sent to the general mailing list, supplemented by a customized patch tracker called Bundle Buggy. Since reviews and the patches themselves are sent to all developers by email, discussion of patches tends to be much more open than Twisted. The large number of patches makes the mailing list more difficult to follow, and it can be easy for review threads to spawn long and winding discussions.

Launchpad's separate mailing list makes it easier to filter general development discussion from code reviews, while allowing third parties to follow other people's reviews. In practice this means that Launchpad draws from the strengths and weaknesses of both approaches.

Real-time or offline?

Doing code reviews asynchronously is very different from doing them in real time.

In asynchronous code reviews, such as ones done over email, the reviewer has time to phrase comments diplomatically. Similarly, the author is able to reply to each point in whichever order they choose.

Real-time code reviews have all the advantages and disadvantages of a natural conversation: misunderstandings can be quickly identified and corrected; there is no long wait for a reply. Of course, it's easier for discussion to become heated and harder to separate emotions from fact. It's also harder to know when you're done. When you receive an email reviewing your code, your mail client will give you some idea of just how long that email is; real-time conversations are open-ended.

Who resolves disputes?

There are times when the author thinks one thing about a patch and the reviewer thinks the opposite, and no reasoning will make the opinions converge. For example, in one patch, the author thought that:

```
[item] = singleton_list
```

was a clear way to get the only value from a list that was guaranteed to have only one value. The reviewer thought that this was unclear and suggested:

```
assert len(singleton_list) == 1, \
    "List should have only one value: %r" % (singleton_list,)
item = singleton_list[0]
```

Despite compelling arguments from both sides, neither would compromise. What should be done? Who resolves the dispute?

Bazaar takes the angle that the patch author has the final say on disputed matters, Twisted says the reviewer has the final say and Launchpad has a head reviewer who has a casting vote on these decisions.

If the author has the final say, then the rate of development will be faster, at the cost of a loss of uniformity and perhaps quality. If the reviewer has the final say, patches will land more slowlyâ€”the reviewer is never as keen to see a patch land as the authorâ€”but the patches will be more rigorously vetted.

Bad things

Ad hominem

This ought to go without saying: review the code and not the coder. Comments about a person will only make it harder for that person to apply critiques about their code.

When making negative comments, refer to “the patch” or “the branch” rather than “you”. For example, “You've introduced a bug in `get_message`” becomes “this patch introduces a bug in `get_message`”.

Unclear Outcomes

If *all* you say is, “this patch introduces a bug in `get_message`”, then you have failed as a reviewer. The goal is to improve the code, not to provide a series of puzzles for the author.

The author should be able to look at a review and be able to tell how to address each point and also when they have addressed all points. Reviews with unclear outcomes turn into open-ended discussions about the patch, which sometimes become focused on making the reviewer happy, rather than improving the code.

Confusing personal preference with objective worth

This is a problem in all spheres of review. Film critics, literary editors and acadamic reviewers all do it. “What I like” is not necessarily the same as “what's good”, although part of becoming a better programmer is having your preferences align better with reality. “What I dislike” is perhaps even less likely to be the same as “what's bad”.

When reviewing a perfectly acceptable patch that solves a problem using imperative-style programming, *do not* criticize it simply because it isn't in a more functional style. Doing otherwise makes reviews a game of “guess what the reviewer likes” rather than “write good code”.

Reviewers can avoid this trap by phrasing review comments as questions, “Did you consider using a more functional style?”, “Why aren't you using regular expressions to solve this problem?” etc.

See also “Specific feedback is good feedback”.

While you're there...

When reviewing code, it is tempting to ask the author to fix other problems that are in the same area. Without care, this can quickly become an exercise in making this area of the code perfect, when previously both reviewer and author were concerned about merely improving the code.

The solution here is to strongly value incremental improvements, see “Better is better, perfect is impossible.”

Filibustering

Filibustering is an American political term for indefinitely prolonging debate on a bill in order to prevent that bill from being enacted. It is sometimes used in Free Software projects to prevent patches from going in.

The effects of filibustering can sometimes happen unintentionally. A reviewer rejects a patch for not having unit tests, the author asks “how do I unit test this code?” and the reviewer never quite gets around to responding.

Apart from being simply an unpleasant process, the author has work left hanging, which makes it harder for them to submit more patches.

See “Unclear outcomes” and “Fast feedback is good feedback”.

Good things

Specific feedback is good feedback

Pinpoint the exact line of code that could be improved. Say what is wrong with it. Suggest one way to make it better. Make sure the author can tell when it will be good enough.

Fast feedback is good feedback

Once authors submit patches for review, there's nothing more they can do on those patches. They have to wait until the reviewer has replied before they can proceed. Reply to reviews quickly to keep the author's attention focused.

Better is better, perfect is impossible

No patch will be perfect, and no patch will fix all problems in existing code. Don't aim for perfection, aim for incremental improvements.

Be thankful

Someone has just tried to improve your product. They've put thought, effort and creativity into helping you, and now they have put their work up for critique: thank them.

Divmod have a policy of always saying one good thing in each code review. There is always something nice to say, even if it's just “I'm glad someone is looking at this part of the code”.

Ask questions

Reviewers can't go wrong with asking questions. In the worst case, the author will get an obvious answer. In the best case, both reviewer and author learn something new.

Conclusion

Code reviews provide an amazing opportunity to grow as a programmer and to improve the software we make. There are many choices that a project can make about how reviews are done and what they can achieve. By thinking carefully about how technologies and processes affect the basic human interactions involved in code review, open source projects can avoid traps that scare off newcomers or wear down longstanding contributors and instead focus on building the best software possible.

Further Reading

- Guido Van Rossum on “Mondrian”, Google's internal code review tool
- The Ultimate Quality Development System
- The Bazaar HACKING Guide
- Beyond copy-editing: the editor-writer relationship