Code Review

BY JOHN BALDWIN

Software is written by imperfect people. As a result, software itself is imperfect as it reflects gaps and misunderstandings in developers' perceptions of both the problem being solved and the software serving as the solution. These gaps and misunderstandings manifest as bugs resulting in incorrect behavior, inconsistent behavior, crashes, data loss, etc. One of the tools to aid in mitigating this is having additional developers review source code. While these developers themselves are also imperfect, the various gaps and misunderstandings will vary from person to person. Hopefully, the more developers who are able to examine a given piece of source code, the more likely it is that bugs in the software can be found. In addition, code review can promote "conceptual integrity" as described by Fred Brooks in *The Mythical Man-Month*. In other words, code review by developers familiar with the overall design of a system can help ensure that new changes are consistent with the system's design, or that changes "do things the way this system normally does them."

Over time, the FreeBSD Project has grown a culture of code review. While code review is not mandatory for changes, the ratio of committed changes that are reviewed continues to increase. When I first became involved with the Project in the late 1990s, code review was fairly informal, especially for changes to the source tree. The changes that were reviewed typically took place in private email threads, over IRC, or, on occasion, in person. During code freezes for releases, code review was somewhat mandated in the form of approval by the release engineering team for code freeze commits. However, this approval was generally focused on a risk assessment of the changes relative to the stability of the upcoming release, rather than on the quality of the code itself. Ports developers have had a stronger review culture than the source tree, and several years ago a couple of ports developers stood up an instance of the Phabricator code review tool to review FreeBSD patches. While the initial focus was on providing a better mechanism than Bugzilla for reviewing patches to ports, several source developers began using it as well.

Tools like Phabricator offer several advantages relative to some of the other systems FreeBSD has used. Reviews often include discussions about various trade-offs or assumptions made by new changes. When these reviews occur in private email threads, that knowledge is only available to the individuals on the private thread. When those reviews occur in Phabricator, those discussions are archived and can be found via a URL from the commit that added the changes to the tree. At the same time, Phabricator permits developers to limit which reviews they wish to participate in without requiring all developers to wade through all of the review comments

on every change. Unlike Bugzilla, Phabricator permits comments on individual lines of a patch. Recent versions of Phabricator include the ability for reviewers to type in code snippets as a suggestion that can be applied to a pending change.

Effective code review requires cooperation both from developers submitting patches for review and developers reviewing changes. The rest of this article will focus on best practices when using FreeBSD's Phabricator instance, though similar guidelines are applicable to reviews over other mediums such as email. Some of these guidelines may also be useful in non-FreeBSD contexts.

When submitting patches, a developer should:

- Upload a patch with full context. Uploading patches using the arcanist tool (from the devel/arcanist port or package) will include this by default. If generating a diff from a version control system, force the diff to contain the full context. When using `diff`, `git diff`, or `git show`, add the `-U999999` argument on the command line. To generate a diff in a subversion checkout, use `svn diff --diff-cmd=diff -x -U999999`. Including full context allows developers to see other portions of the code around a change.
- Provide the commit log for the proposed change in the description of the change. This allows reviewers to review both the code and the commit log describing the change. Writing a proper commit log is a topic for another article.
- Tag relevant groups as reviewers. For example, for a change to the PCI bus drivers, add the #PCI group. Some groups use rules to auto-add groups for files in certain paths. For example, changes to the bhyve hypervisor are automatically tagged with the #bhyve group.
- If the patch submitter has been working with specific developers prior to submitting the patch for review, add those developers as either reviewers or subscribers.
- For FreeBSD source changes, try to format your code based on the guidelines in the style(9) manpage. Some of the guidelines can be ambiguous, and patches do not have to be perfect. However, the closer you are to the existing style, the fewer changes you will have to make later on and the more likely other developers will be willing to review your changes and, if necessary, shepherd them into the tree.
- If a reviewer approves your patch but includes comments stating that the approval is conditional on some changes being applied, then the approval is only valid if the patch submitter makes the requested changes. If, on the other hand, a reviewer approves the patch but includes additional comments that are not required for approval, it is up to the patch submitter to decide whether to make the requested changes. Once a patch has been approved, it can be committed without requiring another upload to Phabricator for explicit approval of requested changes.
- Be patient.
- Be responsive to feedback.

Effective code review requires cooperation both from developers submitting patches for review and developers reviewing changes.

When reviewing changes, developers should:

- Review the design first and before smaller fixes like style. If a change is fundamentally incompatible with the system architecture, it is a waste of everyone's time to require a submitter to restyle the patch only to then reject it on architectural grounds. While some general pointers to style rules may make sense when discussing design, the initial review should focus on the design and structure of a patch.
- If critiquing the design or structure of a patch, provide constructive criticism including alternative designs or methods for achieving the desired outcome.
- Respond in a timely manner. Especially if a reviewer has responded and addressed earlier feedback, it is important to approve the change and/or merge it into the tree.
- Be willing to approve a change that still needs some changes if you trust the submitter to make any needed changes before committing. When requesting changes on such a review, be explicit if further changes are either optional suggestions (so can be committed with or without), or if they are mandatory fixes that must be applied before the approval is valid.
- Be willing to recognize that some suggestions are really preferences and categorize them as such rather than as mandatory fixes.
- When noticing a pattern of necessary changes (e.g., a style rule that the submitter might have misunderstood or not known), rather than pointing out every flaw on the first review, point out a single instance with a reference to the general rule and let the submitter have a chance to resolve the issue throughout the patch.

JOHN BALDWIN is a systems software developer. He has directly committed changes to the FreeBSD operating system for 20 years across various parts of the kernel (including x86 platform support, SMP, various device drivers, and the virtual memory subsystem) and userspace programs. In addition to writing code, John has served on the FreeBSD core and release engineering teams. He has also contributed to the GDB debugger and LLVM. John lives in Concord, California, with his wife, Kimberly, and three children: Janelle, Evan, and Bella.