Lightweight Code Review Episode 3:
Pros and Cons of Four Kinds of Code Review

In Episode 2 we explained why the venerable Formal Inspection technique isn't used in practice and why a "lightweight" style of review might be useful. In this Episode we explore the pros and cons of four other common styles of code review and see which one is the most promising candidate for effective but not time-consuming practical peer code reviews.

Skinning Cats

There are many ways to skin a cat. I can think of four right off the bat. There are also many ways to perform a peer review, each with pros and cons. I hope this turns out to be a bad analogy...

We've already explored why the tried-and-true Formal Inspection method of code review doesn't work in practice. Several other, simpler techniques suggest themselves as alternatives:

- **Over-the-shoulder:** One developer looks over the author's shoulder as the latter walks through the code.
- **Email pass-around:** The author (or SCM system) emails code to reviewers.
- **Pair Programming:** Two authors develop code together at the same workstation.
- **Tool-assisted:** Authors and reviewers use specialized tools designed for peer code review.

Over-the-shoulder reviews

This is the most common and informal (and easiest!) of code reviews. An "over-the-shoulder" review is just that -- a developer standing over the author's workstation while the author walks the reviewer through a set of code changes.

Typically the author "drives" the review by sitting at the keyboard and mouse, opening various files, pointing out the changes and explaining why it was done this way. The author can present the changes using various tools and even run back and forth between changes and other files in the project. If the reviewer sees something amiss, they can engage in a little "spot pair-programming" as the author writes the fix while the reviewer hovers. Bigger changes where the reviewer doesn't need to be involved are taken off-line.

With modern desktop-sharing software a so-called "over-the-shoulder" review can be made to work over long distances, although this can complicate the process because you need to schedule these sharing meetings and communicate over the phone.

The most obvious advantage of over-the-shoulder reviews is simplicity in execution. Anyone can do it, any time, without training. It can also be deployed whenever you need it most -- an especially complicated change or an alteration to a "stable" code branch.

Before I list out the pros and cons, I'd like you to consider a certain effect that only this type of review exhibits. Because the author is controlling the pace of the review, often the reviewer doesn't get a chance to do a good job. The reviewer might not be given enough time to ponder a complex portion of code. The reviewer doesn't get a chance to poke around other source files to check for side-effects or verify that API's are being used correctly.

The author might explain something that clarifies the code to the reviewer, but the next developer who reads that code won't have the advantage of that explanation unless it is encoded as a comment in the code. It's difficult for a reviewer to be objective and aware of these issues while being driven through the code with an expectant developer peering up at him.

So:

- Pro: Easy to implement
- Pro: Fast to complete
- Pro: Might work remotely with desktop-sharing and conference calls
- Con: Reviewer led through code at author's pace
- Con: Usually no verification that defects are really fixed
- Con: Easy to accidentally skip over a changed file
- Con: Impossible to enforce the process
- Con: No metrics or process measurement/improvement

Email pass-around reviews

This is the second-most common form of lightweight code review, and the technique preferred by most open-source projects. Here, whole files or changes are packaged up by the author and sent to reviewers via email. Reviewers examine the files, ask questions and discuss with the author and other developers, and suggest changes.

The hardest part of the email pass-around is in finding and collecting the files under review. On the author's end, he has to figure out how to gather the files together. For example, if this is a review of changes being proposed to check into version control, the user has to identify all the files added, deleted, and modified, copy them somewhere, then download the previous versions of those files (so reviewers can see what was changed), and organize the files so the reviewers know which files should be compared with which others. On the reviewing end, reviewers have to extract those files from the email and generate differences between each.

The version control system can assist the process by sending the emails out automatically. The automation is helpful, but for many code review processes you want to require reviews *before* check-in, not *after*.

Like over-the-shoulder reviews, email pass-arounds are fairly easy to implement. They also work just as well across the hall or across an ocean.

A unique advantage of email-based review is the ease in which other people can be brought into conversations, whether for expert advice or complete deferral. And unlike over-the-shoulder, emails don't break developers out of "the zone" as they are working; reviews can be done whenever the reviewer has a chance.

The biggest drawback to email-based reviews is that they can quickly become an unreadable mass of comments, replies, and code snippets, especially when others are invited to talk and with several discussions in different parts of the code. It's also hard to manage multiple reviews at the same time. Imagine a developer in Hyderabad opening Outlook to discover 25 emails from different people discussing aspects of three different code changes he's made over the last few days. It will take a while just to dig though that before any real work can begin.

Another problem is that there's no indication that the review is "done." Emails can fly around for any length of time.

So:

- Pro: Fairly easy to implement
- Pro: Works with remote developers
- Pro: SCM system can initiate reviews automatically
- Pro: Easy to involve other people
- Pro: Doesn't interrupt reviewers
- Con: Usually no verification that defects are really fixed
- Con: How do you know when the review is "complete?"
- Con: Impossible to know if reviewers are just deleting those emails
- Con: No metrics or process measurement/improvement

Pair-programming (review)

Most people associate pair-programming with XP and agile development in general.

Among other things, it's a development process that incorporates continuous code review. Pair-programming is two developers writing code at a single workstation with only one developer typing at a time and continuous free-form discussion and review.

Studies of pair-programming have shown it to be very effective at both finding bugs and promoting knowledge transfer. And some developers really enjoy doing it. (Or did you forget that making your developers happy is important?)

There's a controversial issue about whether pair-programming reviews are better, worse, or complementary to more standard reviews. The reviewing developer is deeply involved in the code, giving great thought to the issues and consequences arising from different implementations. On the one hand, this gives the reviewer lots of inspection time and a deep insight into the problem at hand, so perhaps this means the review is more effective. On the other hand, this closeness is exactly what you don't want in a reviewer; just as no author can see all typos in his own writing, a reviewer too close to the code cannot step back and critique it from a fresh and unbiased position. Some people suggest using both techniques -- pair-programming for the deep review and a follow-up standard review for fresh eyes. Although this takes a lot of developer time to implement, it would seem that this technique would find the greatest number of defects. We've never seen anyone do this in practice.

The single biggest complaint about pair-programming is that it takes too much time. Rather than having a reviewer spend 15-30 minutes reviewing a change that took one developer a few days to make, in pair-programming you have two developers on the task the entire time.

Of course pair-programming has other benefits, but a full discussion of this is beyond the scope of this article and is already discussed all over the Internet.

So:

- Pro: Shown to be effective at finding bugs and promoting knowledge-transfer

- Pro: Reviewer is "up close" to the code so can provide detailed review
- Pro: Some developers like it
- Con: Some developers don't like it
- Con: Reviewer is "too close" to the code to step back and see problems
- Con: Consumes a lot of up-front time
- Con: Doesn't work with remote developers
- Con: No metrics or process measurement/improvement

Tool-assisted review

This refers to any process where specialized tools are used in all aspects of the review: collecting files, transmitting and displaying files, commentary, and defects among all participants, collecting metrics, and giving product managers and administrators some control over the workflow.

"Tool-assisted" means you either bought a tool (like ours) or you wrote your own. Either way, this means money -- you're either paying us for the tool or paying your own folks to create and maintain it. Plus you have to make sure the tool matches your desired workflow, and not the other way around.

Therefore, the tool had better provide Many Advantages if it is to be worthwhile. Specifically, it needs to fix the major problems of the foregoing types of review with:

- **Automated File-Gathering:** As we discussed in email pass-around, developers shouldn't be wasting their time collecting "files I've changed" and all the differences. Ideally the tool should be able to collect changes *before* they are checked into version control *or after*.
- **Combined Display: Differences, Comments, Defects:** One of the biggest time-sinks with any type of review is in reviewers and developers having to associate each sub-conversation with a particular file and line number. The tool must be able to display files and before/after file differences in such a manner that conversations are threaded and no one has to spend time

cross-referencing comments, defects, and source code.

- **Automated Metrics Collection:** On one hand, accurate metrics are the only way to understand your process and the only way to measure the changes that occur when you change the process. On the other hand, no developer wants to review code while holding a stopwatch and wielding line-counting tools.

  A tool that automates the collection of key metrics is the only way to keep developers happy (i.e., no extra work for them) and get meaningful metrics on your process. A full discussion of review metrics and what they mean (and don't mean) will appear in another article, but your tool should at least collect these three things: kLOC/hour (inspection rate), defects/hour (defect rate), and defects/kLOC (defect density).
- **Workflow Enforcement:** Almost all other types of review suffer from the problem of product managers not knowing whether developers are reviewing all code changes or whether reviewers are verifying that defects are indeed fixed and didn't cause new defects. A tool should be able to enforce this workflow at least at a reporting level (for passive workflow enforcement) and at best at the version control level (with server-side triggers).
- **Clients and Integrations:** Some developers like command-line tools. Others need integrations with IDE's and version control GUI clients. Administrators like zero-installation web clients and Web Services API's. It's important that a tool supports many ways to read and write data in the system.

If your tool satisfies this list of requirements, you'll have the benefits of email pass-around reviews (works with multiple, possibly-remote developers, minimizes interruptions) but without the problems of no workflow enforcement, no metrics, and wasting time with file/difference packaging, delivery, and inspection.

It's impossible to give a proper list of pros and cons for tool-assisted reviews because it depends on the tool's features. But if the tool satisfies all the requirements above, it should be able to combat all the "cons" above.

So what do I do?

All of the techniques above are useful and will result in better code than you would otherwise have. All of our experience with peer code review has been poured into articles and books and Code Collaborator -- software for tool-assisted code reviews.

We have data that simultaneously shows how to maximize peer review effectiveness while minimizing amount of time spent doing it. This is the subject of our next article.

What's Next: *In the next article we'll provide hard data to back up our own theory of tool-assisted lightweight code review. Later in the series we'll explore often-overlooked aspects of review such as dealing with the social ramifications of personal critiques.*

By Jason Cohen (jason.cohen@smartbearsoftware.com)