Reviewing Code with Perforce

Jason Cohen, Smart Bear Software

Executive Summary

Code review is an efficient and cost-effective way to find software defects and improve code quality. Fortunately, Perforce is designed to facilitate peer code review. Combining features in Perforce with modern techniques, it's easy and fast to do. This paper describes several ways to conduct code reviews with Perforce and examines the implications of each. It also discusses best practices and considerations for reviewing code as quickly, painlessly and successfully as possible.

Why Review Code?

Most programmers detest code reviews, especially formal inspections in the traditional sense. Reviews have the bad rap of taking an inordinate amount of time, and they take time away from task programmers enjoy most… writing code! At the same time, most folks also know that the benefits of code review are undeniable and well-proven. If you think through the concept, common sense tells you how important reviews are.

You wouldn't write a book or even an article without having someone else review it – usually several people. Why would you even consider writing code without at least one extra pair of eyeballs to do a sanity check?!

The most obvious result of having someone else look at your code is *better software quality*. You won't just ship fewer bugs, you'll improve design, structure, documentation, maintainability, testing, and even comments. But code review provides a number of other important benefits too:

- *It improves team communication*. Code review gets people talking, collaborating, and building trust in addition to better code. Once the "ice is broken," many teams report vast improvements in how they work together.

- *Code review provides a safe opportunity to mentor and teach junior team members.* Newer folks can still work on 'real' projects, and the team can be sure they don't do any serious damage. Code review is also a wonderful way for new folks to learn about your code base.

- *It propagates code expertise*. Code review increases your whole team's knowledge about the entire code base. The alternative – having only one or two team members who are experts on all (or even parts) of your code – is a huge risk.

However, the purpose of this paper is not to convince you to review code, but rather to explain how it's done in Perforce and to outline the implications and pros/cons of the various possible processes.

Methods of Code Review Using Perforce

You have a choice of multiple good methods to review code using Perforce. Let's talk about how they work so you can pick the one that works best for your team.

Over the Shoulder

In *Over the Shoulder* reviews, one person stands behind another and the author steps through his or her code with the reviewer. When the team finds bugs, the bugs are fixed on the spot or entered into an issue-tracking program.

In Perforce, this kind of review is easily accomplished using the built-in "Diff" viewer P4Merge. Typically you'll right-click on a changelist in P4V and examine the changes between the current and previous versions of the files in that changelist. This approach works equally well with pending or submitting changelists, although with the former the differences have to be created on the author's workstation, whereas with the latter you can extract diffs at any time from anyone's workstation.

This informal method works well when developers are in the same office and can schedule time to work together. It falls apart when developers are geographically-separated or have a lot of meetings or scheduling conflicts to work around.

Quick Pros and Cons:

- 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

Meeting-Based Walk-Through

Meeting-based Walk-Throughs are like *Over the Shoulder* reviews, except that teams get together and review code on a projector, using a virtual meeting, or with code printed out. The Perforce approach is the same as with the *Over the Shoulder* method.

Quick Pros and Cons:

- Involves the same issues as *Over the Shoulder* reviews
- Con: Requires additional time for scheduling/waiting overhead
- Con: The next developer who has to maintain this code won't have the benefit of it being explained to them by the author. So if the information isn't *in the code*, it's still a problem. In meetings-based walk-throughs, you usually don't consider this issue, nor are you really in a position to evaluate it since you're not really reading the code yourself.

The traditional method of code review, meeting-based reviews are the least popular. Everyone must make time in their schedules (whether at a convenient time or not) to attend meetings, an interruption that often pulls programmers out of "the zone." Having too many meetings tends to cause universal resentment. And meetings can drag on if not well-controlled, especially when more than 2-3 people attend. This method of code review is sometimes too time-consuming to be practical.

Email Pass-Around

In *Email Pass-Around* reviews, team members can email each other code as it's developed and send comments back and forth. 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 review consists 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. And everyone has to manually reference line numbers in their emails and on the code.

Perforce 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.

This informal method works pretty well for geographically-distributed teams, but lacks the ability to track issues or tie comments to the specific section of code they apply to. And email threads can rapidly become long and confusing. 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.

Quick Pros and Cons:

- 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

Tool-Supported

The fastest and most efficient method, tool-supported code reviews use dedicated software to automate uploads and facilitate online discussions. Some tools allow comments directly on the code itself at the relevant location, so you don't have to search for line numbers. Tools also include workflow to keep the process moving along, issue-tracking capabilities, and other time-saving functions. Tool-assisted reviews take about 1/5th of the time of full-on formal reviews (meetings) because the tools handle the tedious parts of the review. They work well for both local and distributed teams.

Tools should include the following functionality:

- Automated File-Gathering
- Combined Display: Differences, Comments, Defects
- Automated Metrics Collection
- Workflow Enforcement
- Clients and Integration

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. Code Collaborator™ and Crucible are the most well-known commercially-available tools. Review Board is an option if you want an open-source tool.

Quick Pros and Cons:

- 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" of the other methods.

World's Largest Code Review Case Study: Cisco, Perforce, & Code Review Tool

Cisco Systems worked with my company, Smart Bear Software, to conduct the world's largest published case study of peer code review, spanning 2500 reviews of 3.2 million lines of code over a 10-month period. We used Perforce and our Code Collaborator code review tool in the study, which was designed to examine how programmers review code and how to improve the process.

At the start of the study, we set up some rules for the group:

- All code had to be reviewed before it was checked into the team's Perforce version control software.
- Smart Bear's Code Collaborator code review software tool would be used to expedite, organize, and facilitate all code review.
- In-person meetings for code review were not allowed.
- The review process would be enforced by tools.
- Metrics would be automatically collected by Code Collaborator, which provides review-level and summary-level reporting.

The case study yielded process implications and best practices for any kind of review.

Review for No Longer than 60-90 Minutes

Reviews should not last longer than 60-90 minutes. After that amount of time, programmers get tired, lose focus, and performance starts to drop off.

This conclusion is well-supported by evidence from many other studies besides our own. In fact, it's generally known that when people engage in any activity requiring concentrated effort, performance starts dropping off after 60-90 minutes. On the flip side, you should always spend at least five minutes reviewing code – even if it's just one line. Often a single line can have consequences throughout the system, and it's worth the five minutes to think through the possible effects a change can have.

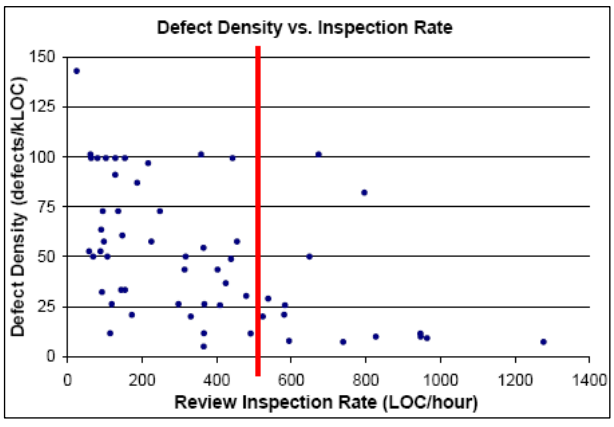

Figure 1: Inspection effectiveness falls off when greater than 500 lines of code are under review.

Optimal Velocity: 300-500 LOC Per Hour or Less

Take your time with code review. Faster is not better. Our case study showed that you'll achieve optimal results at an inspection rate of less than 300-500 lines of code (LOC) per hour. Reviewing faster than 400-500 LOC/hour results in a severe drop-off in effectiveness. And at rates above 1000 LOC/hour, you can probably conclude that the reviewer isn't actually looking at the code at all.

It makes sense that if you don't spend much time on the review or you go too quickly, you won't find most of the bugs that lurk there. If you try to review too much code at once, you won't be able to devote enough detailed attention to the code you're reviewing, so you won't catch the bugs.

Review Fewer than 200-400 LOC at Once

The Cisco code review study showed that for optimal effectiveness, developers should review fewer than 200-400 LOC at a time. Beyond that, your ability to find defects diminishes. At this rate, with the review spread over no more than 60-90 minutes, you should get a 70-90% yield; in other words, if 10 defects existed, you'd find 7-9 of them.

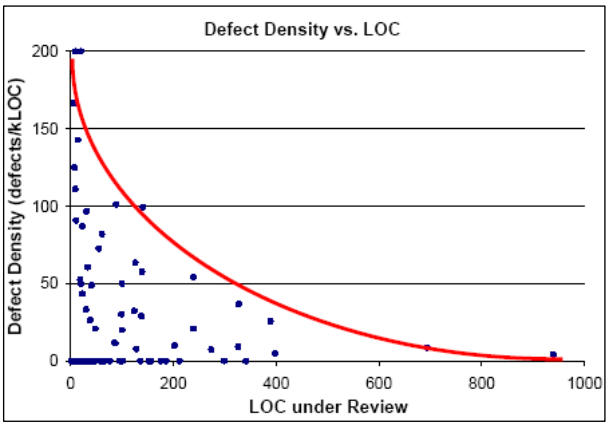

Figure 2: Defect density dramatically decreases when the number of lines of inspection goes above 200, and is almost zero after 400.

Author Preparation Matters – A Lot

Another best practice we explored is that author preparation makes a huge difference in code quality and review efficiency.

It occurred to us that authors might be able to eliminate most defects before a review even begins. If we required developers to double-check their work, maybe reviews could be completed faster without compromising code quality. As far as we could tell, this idea specifically had not been studied before, so we tested it during the study at Cisco.

The concept behind "author preparation" is that authors should annotate their source code before the review begins. Annotations guide the reviewer through the changes, showing which files to look at first and defending the reason and methods behind each code modification. These notes are not comments in the code, but rather comments given to other reviewers.

Author preparation provides two huge benefits. First (and most obviously), the annotations help the reviewer know which files to look at and explain the reasoning behind methods chosen. But the second benefit is even more powerful: often, as authors are re-thinking and explaining their code, they discover their own defects before the review even begins, thus making the review itself more efficient. Of course, this kind of self-review is a good idea even if your team doesn't do code reviews!

These best practices represent only a small portion of what we learned in the case study. If you want more details, our joint findings are outlined in the book, *Best Kept Secrets of Peer Code Review*, available *free* at www.CodeReviewBook.com.

Getting Started with Code Review: Start Small for Most Powerful Results

Now you know about methods of code review with Perforce and you've learned some best practices. So what's the best way get started?

The optimal way to begin is at a small scale, by reviewing only a handful of files. Don't plan to review all of your code at the outset. Instead, these suggestions for limiting review scope will help you achieve maximum results in minimal time. Choose however many of them make sense for your team.

- Review changes to the stable branch only
- Review changes to the core module that all other code depends on
- Review changes to the "top 10 scariest files" as voted by the developers
- Review only unit tests – if they are complete and not over-specified, any bugs or other problems can be fixed safely later
- Review only changes, not entire files
- Review code whenever developers think it's necessary, such as when they have concerns about a section of code, when a domain expert can lend specific expertise, or when they're working on code they didn't write and may not fully understand.

Starting small and focusing on the most important files is the most efficient review you can do, and it shows the most obvious results for minimal time spent… which yields another advantage. When programmers see dramatic results that improve their code but require little effort on their party, they buy into the process! Implementing code review *without* team buy-in almost pre-destines the process to fail. Rather, the team needs to recognize the benefits and agree to the process in order to be motivated and actually *do* reviews. Starting small helps convince skeptics that code review is worth everyone's time.

Additional Review Considerations and Best Practices

Branching Policy

Many folks who do code review have voluntarily changed their branching habits to reduce developer down-time. You review code, sometimes before check-in and sometimes after, but at any rate you should be able to just layer it on top of your current branching scheme. And you can. Often there''s nothing more to it, especially if you want to review code before check-in. But requiring a review before code is committed to version control has a price: What does the author do while waiting for the review to complete?

The right branching policy can save the day here. Instead of requiring "review before check-in," require that changes be made in a "local sandbox," meaning a little branch made for one developer for the purpose of working on a single task. When the task is done, review the branch, rather than reviewing the "changes not yet checked in." To "review the branch" in Perforce, the easiest way is to start an "integration" changelist, resolve merge conflicts, and then perform the review on the result (before you check in it!).

Once the review is set up, the developer can start working on something else in a different branch. If the new task is unrelated, resuming coding poses no problem. If the new task depends on the changes from the first task, the new branch can be based on the first branch. That way, if fixes come back, they can be applied directly (and re-synched with the new task), but if no problems are found then the new task proceeds unhindered.

Local sandboxes have other desirable features besides un-blocking early-stage code review. The developer can now use Perforce for intermediate steps rather than having to wait for days before a large check-in can occur.

You can also have more than one developer using a local sandbox. Perhaps two or three developers are working together on a new feature, or perhaps a Scrum group shares a sandbox until they are nearing the end of the iteration and want to do code reviews with each other as part of the definition of "done."

The downside to branching is the cost of having more branches on the Perforce server. To some extent, this issue can be managed by pruning old branches and detecting when a branch has been started but abandoned, but that means more work

that someone has to do. Still, the Perforce administration screens in P4V make it easy to visualize branches, so it's possible to identify things that look abandoned.

In the end, the purpose of your IT infrastructure is to support software development. Developers should not be unreasonably blocked from writing code. The cost of upgrading some servers is worth it.

Checklists

Another simple, highly-recommended way to improve code quality is to use checklists to find the things you forget to do. They're useful for both authors and reviewers. Consider that omissions are the hardest defects to find – after all, it's hard to review something that's not there. A checklist is the single best way to combat the problem, as it reminds the reviewer or author to take the time to look for something that might be missing. A checklist reminds authors and reviewers to confirm that all errors are handled, that function arguments are tested for invalid values, and that unit tests have been created.

Another useful concept is the personal checklist. Each person typically makes the same 15-20 mistakes. If you notice what your typical errors are, you can develop your own personal checklist (PSP, SEI, and CMMI recommend this practice too). Reviewers will do the work of determining your common mistakes. All you have to do is keep a short checklist of the 5-10 most common flaws in your work, particularly the things you forget to do. As soon as you start recording your defects in a checklist, you will start making fewer of them. The rules will be fresh in your mind and your error rate will drop.

Checklists are especially important for reviewers, since if the author forgot it, the reviewer is likely to miss it as well.

For more detailed information on how to build checklists, read this whitepaper: http://tinyurl.com/cypvoy.

Conclusion

Code review will significantly improve the quality of your software, and Perforce provides a variety of features and workflow that support streamlined code reviews. It's important to choose a method and a process that works well for your team.

"Still… is it really worth our time?" Good question. The easiest way to find out is to try it and measure the actual results for your team. Get your team to consent to try code review for one week and to agree to spend a total of 2-2.5 hours per programmer. Then follow these steps:

1) Establish the code review method you're going to use. Tool-supported, over-the shoulder, or email pass-around methods are recommended for this evaluation week. These methods are easiest to conduct and track.

2) Have everyone do code reviews for 20-30 minutes per day for that one week.

3) During each review, capture two simple metrics: Number of Bugs Found and Time Spent.

4) Calculate the rate at which your team finds bugs. Divide the total time spent on reviews by the number of defects found. For example: 25 hours spent divided by 100 defects found yields 0.25 hour (15 min) spent to find one defect.

Consider how more much time and money that defect would cost to fix if it slipped through to QA (typically 8-12X), or worse, to customers (30-100X!). Now discuss the results with the team and let the group decide… is it worth it?

If the team doesn't see the benefits at that point, then code review is not for you. But chances are they will, and then you'll have the buy-in you need to conduct successful code reviews! And now you know how to do them right.

About the Author

Jason Cohen is the founder of Smart Bear Software (www.SmartBear.com). In conjunction with Cisco Systems, Jason conducted the world's largest study of lightweight peer code review and published the findings in his book, Best Kept Secrets of Peer Code Review.