Lightweight Code Review Episode 4: The Largest Case Study of Code Review, Ever

In Episode 3 we discussed several types of code review. In this Episode we give the results of the largest-ever case study on peer review.

Forget theory -- what's it really like?

The ACM and IEEE archives are replete with papers on code review and inspection. But after a casual perusal you notice that most studies are done with fewer than 50 reviews, usually in a university setting with students and contrived code samples.

That's fine for academics, but what's it like in the real world? With real software developers (ranging from junior to senior) and real software projects with 1000's of files and real deadlines?

2500 reviews. 50 developers. Real software

In May 2006 we wrapped up the largest case study of peer code review ever published, done at Cisco Systems&reg;. The software was MeetingPlace&reg; -- Cisco's computer-based audio and video teleconferencing solution. Over 10 months, 50 developers on three continents reviewed every code change before it was checked into version control.

We collected data from 2500 reviews of a total of 3.2 million lines of code. This article summarizes our findings.

How reviews were conducted

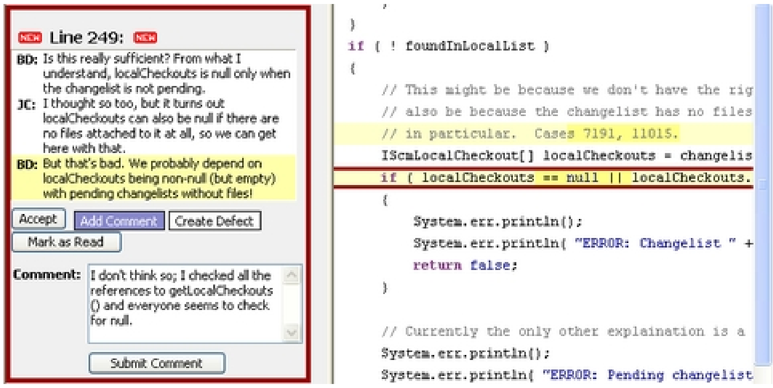
The reviews were conducted using Smart Bear Software's Code Collaborator system for tool-assisted peer review. This article is not intended to be a sales pitch for Collaborator, so please see the website for product details.

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

Cisco wanted to review all changes before they were checked into the version control server, which in their case was Perforce&reg;. They used a Perforce server-side trigger (part of Code Collaborator) to enforce this rule.

Developers were provided with several Code Collaborator tools which allowed them to upload local changes from a command-line, from a Windows GUI application, and within the Perforce GUI applications P4Win and P4V.

Reviews were performed using Code Collaborator's web-based user interface:



BD: Is this really sufficient? From what I understand, localCheckouts is null only when the changelist is not pending.

JC: I thought so too, but it turns out localCheckouts can also be null if there are no files attached to it at all, so we can get here with that.

BD: But that's bad. We probably depend on localCheckouts being non-null (but empty) with pending changelists without files!

Comment: I don't think so; I checked all the references to getLocalCheckouts () and everyone seems to check for null.

```
}
if ( ! foundInLocalList )
{
    // This might be because we don't have the rig
    // also be because the changelist has no files
    // in particular.  Cases 7191, 11015.
    IScmLocalCheckout[] localCheckouts = changelis
    if ( localCheckouts == null || localCheckouts.
    {
        System.err.println();
        System.err.println( "ERROR: Changelist " +
        return false;
    }

    // Currently the only other explaination is a
    System.err.println();
    System.err.println( "ERROR: Pending changelist
```

The Code Collaborator software displayed before/after side-by-side views of the source code under inspection with differences highlighted in color. Everyone could comment by clicking on a line of code and typing. As shown above, conversations and defects are threaded by file and line number.

Defects were logged like comments but tracked separately by the system for later reporting and to create a defect log automatically. Cisco configured the system to collect severity and type data for each defect.

If defects were found, the author would have to fix the problems and re-upload the files for verification. Only when all reviewers agreed that no more defects existed (and previously found defects were fixed) would be review be complete and the author allowed to check in the changes.

Code Collaborator collected process metrics automatically. Number of lines of code, amount of person-hours spent in the review, and number of defects found were all recorded by the tool (no stopwatch required). Reports were created internally for the group and used externally by Smart Bear to produce the analysis for the case study.

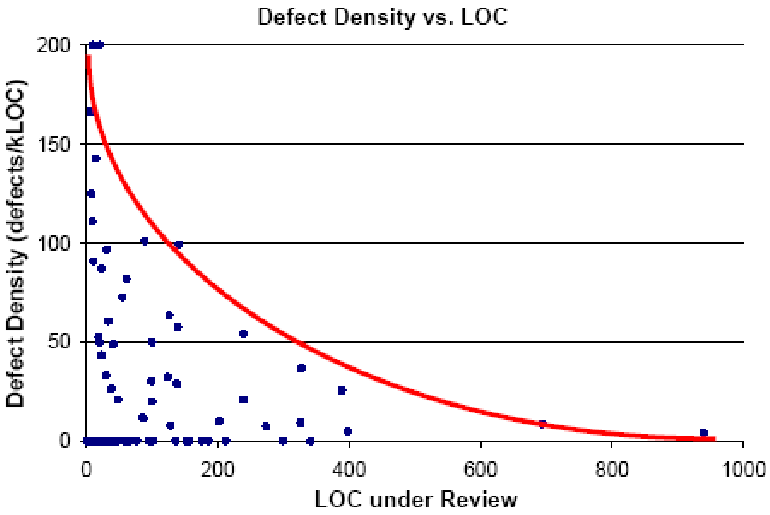
Jumping to the end of the story

The reader will no doubt find it disturbing that, after setting up the parameters for the experiment, we suddenly present conclusions without explanation of statistical methods, the handling of experimental control issues, identifying "defects" that weren't logged as such, and so forth.

The length of this article prevents a proper treatment of the data. The patient reader is referred to Chapter 5 of *Best Kept Secrets of Peer Code Review* for a detailed account.

Conclusion #1: Don't review too much code at once (&lt;200-400 LOC)

As the chart below indicates, defect density decreased dramatically when the number of lines of code under inspection went above 200:



By "defect density" we mean the number of defects found per amount of code, typically per 1000 lines of code as shown on this graph. Typically you expect at least 50 defects per kLOC for new code, perhaps 20-30 for mature code. Of course these types of "rules" are easily invalidated depending on the language, the type of development, the goals of the software, and so forth. A future article will discuss the interpretation of such metrics in more detail.

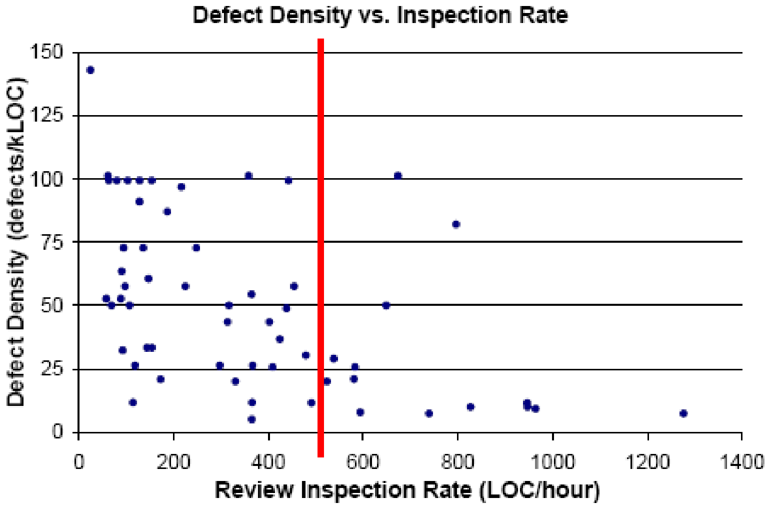
In this case, think of defect density as a measure of "review effectiveness." Here's an example to see how this works. Say there are two reviewers looking at the same code. Reviewer #1 finds more defects than reviewer #2. We could say that reviewer #1 was "more effective" than reviewer #2, and the number of defects found is a decent measure of exactly how effective. To apply this logic across different reviews you need to normalize the "number of defects" in some way -- you'd naturally expect to find many more defects in 1000 lines of code than in 100.

So this chart tells us that as we put more and more code in front of a reviewer, her effectiveness at finding defects drops. This is sensible -- the reviewer doesn't want to spend weeks doing the review, so inevitably she won't do as good a job on each file.

This conclusion may seem obvious, but this data shows exactly where the boundary is between "OK" and "too much." 200 LOC is a good limit; 400 is the absolute maximum.

Conclusion #2: Take your time (&lt;500 LOC/hour)

This time we compare defect density with *how fast* the reviewer went through the code.



Again, the general result is not surprising: If you don't spend enough time on the review, you won't find many defects.

The interesting part is answering the question "How fast is too fast?" The answer is that 400-500 LOC/hour is about as fast as anyone should go. And at rates above 1000 LOC/hour, you can probably conclude that the reviewer isn't actually looking at the code at all.

Conclusion #3: Spend less than 60 minutes reviewing

Let's combine the two previous conclusions. If we shouldn't review more than 400 LOC at once, and if we shouldn't review faster than 400 LOC per hour, then we shouldn't review for more than one hour at a time.

This conclusion is well-supported not only by our own evidence but that from many other studies. In fact, it's generally known that when people engage in *any* activity requiring concentrated effort, performance starts dropping off after 60-90 minutes.

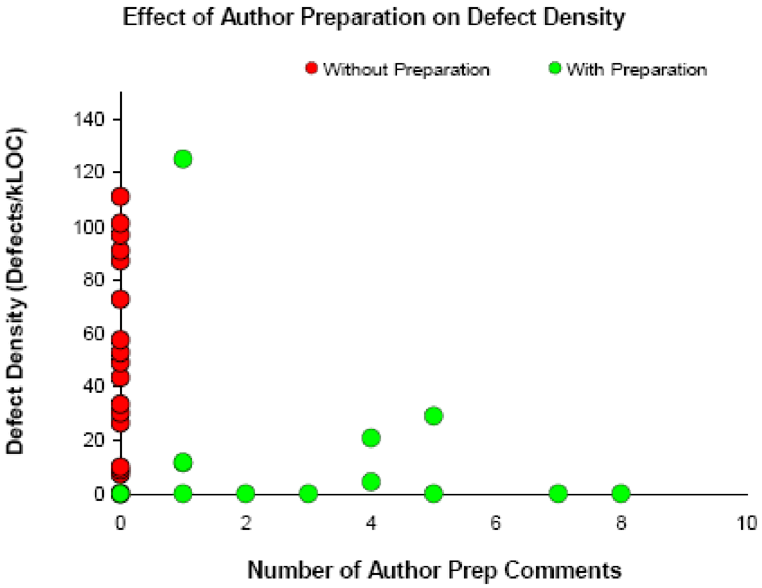
Space does not permit us to delve into this fascinating subject, but this (correct) conclusion derived from the other data helps to support the credibility of the entire study.

Conclusion #4: Author preparation results in more efficient reviews

"Author preparation" is a term we invented to describe a certain behavior pattern we saw during the study. The results of this behavior were completely unexpected and, as far as we know, have not been studied before.

"Author preparation" is when the author of the code under review annotates his code with his own commentary before the official review begins. These are not comments in the code, but rather comments given to other reviewers. About 15% of the reviews in this study exhibited this behavior.

The striking effect of this behavior is shown in this chart showing the effect of author preparation on defect density:



Reviews with author preparation have barely any defects compared to reviews without author preparation.

Before drawing any conclusions, however, consider that there are at least two diametrically opposite ways of interpreting this data.

The *optimistic* interpretation is: During author preparation the author is retracing his steps and explaining himself to a peer. During this process the author will often find defects all by himself. This "self review" results in fixes even before other reviewers get to the table. So the reason we find few defects in author-prepared reviews is that the author has already found them! Terrific!

The *pessimistic* interpretation is: When the author makes commentary the reviewer becomes biased. If the author says "Here's where I call foo()," the reviewer simply verifies that indeed the author called foo() but doesn't ask the deeper questions like "Should we even be calling foo()?" The review becomes a matching game instead of a thoughtful exercise. So the reason we find few defects in author-prepared reviews is that the reviewers have switched off their minds! Terrible!

We resolved this by taking a random sample of 300 reviews that contained author preparation. We examined each one looking for evidence to support one hypothesis or the other.

Our findings were clearly in favor of the optimistic interpretation. There were many cases where reviewers argued with the preparatory comments and where reviewers clearly were looking at other parts of the code and thinking about ramifications elsewhere in the code base.

So, our conclusion is that author preparation is indeed a Good Thing and that it saves time overall during the review because reviewers are not having to call out "obvious" defects.

Results in summary

To summarize all our results, including some things not discussed here:

- Lightweight-style reviews are effective and efficient
-

Review fewer than 200-400 LOC at a time

- Aim for an inspection rate of less than 300-500 LOC/hour
- Take enough time for a proper, slow review, but not more than 60-90 minutes
- Author preparation is good
- Expect defect rates around 15/hour, higher only when &lt;150 LOC under review
- Left to their own devices, reviewers' inspection rates will vary widely, even with similar authors, reviewers, files, and review size

For a full explanation and evidence for all these results, refer to *Best Kept Secrets of Peer Code Review*.

What's Next: *In the next few articles we'll explore often overlooked aspects of review such as dealing with the social ramifications of personal critiques and what metrics really tell us.*

Cisco&reg; and MeetingPlace&reg; are registered trademarks of Cisco Systems Inc.. These names and the information herein are reproduced with permission.

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