Lightweight Code Review Episode 1: The Case for Code Review

It was only supposed to take an hour.

The bad news was that we had a stack of customer complaints. The latest release had a nasty bug that slipped through QA. The good news was that some of those complaints included descriptions of the problem -- an unexpected error dialog box -- and one report had an attached log file. We just had to reproduce the problem using the log and add this case to the unit tests. Turn around a quick release from the stable branch and we're golden.

Of course that's not how it turned out. We followed the steps from the log and everything worked fine. QA couldn't reproduce the problem either. Then it turned out the error dialog was a red herring -- the real error happened long before the dialog popped up, somewhere deep in the code.

A week later with two developers on the task we finally discovered the cause of the problem. Once we saw the code it was painfully obvious -- a certain subroutine didn't check for invalid input. By the time we got the fix out we had twenty more complaints. One potential customer that was trialing the product was never heard from again.

All over a simple bug. Even a cursory glance over the source code would have prevented the wasted time and lost customers.

The worst part is that this isn't an isolated incident. It happens in all development shops. The good news? A policy of peer code review can stop these problems at the earliest stages, before they reach the customer, before it gets expensive.

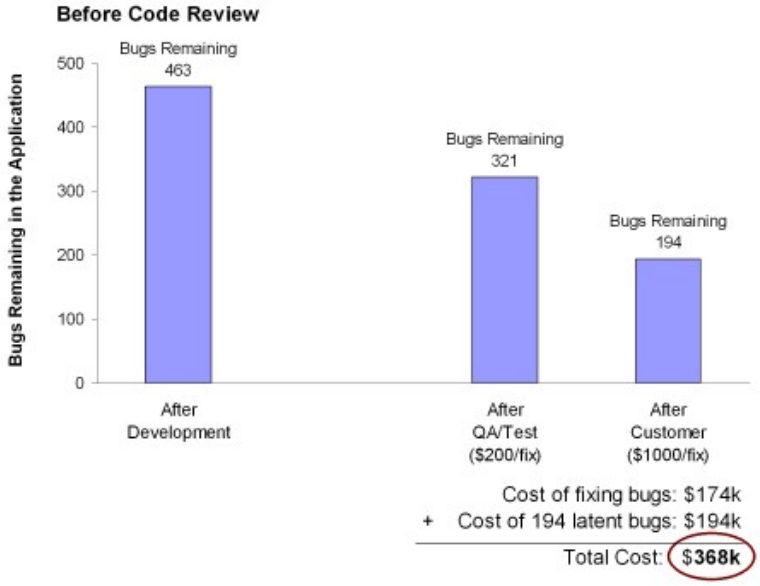
The case for review: Find bugs early & often

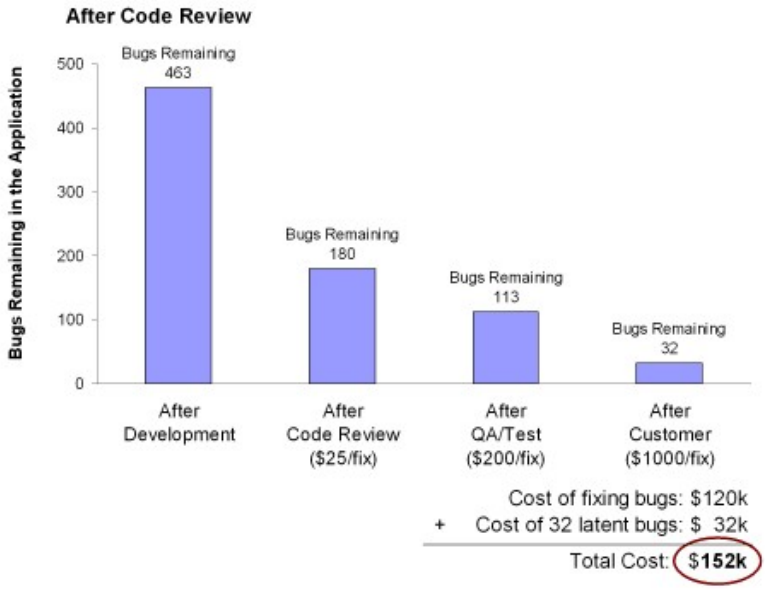
One of our customers set out to test exactly how much money the company would have saved had they used peer review in a certain three-month, 10,000-line project with 10 developers. They tracked how many bugs were found by QA and customers in the subsequent six months. Then they went back and had another group of developers peer-review the code in question. Using metrics from previous releases of this project they knew the average cost of fixing a defect at each phase of development, so they were able to measure directly how much money they would have saved.

The result: Code review would have saved *half* the cost of fixing the bugs. Plus they would have found 162 additional bugs.

Examine the following charts for the raw data.

Saving $150k: A real-world case study





(Critical readers might take issue with the given dollar amounts. These numbers came from the customer and of course they are arguable and differ between organizations. However, as long as you agree that "bugs found later in the process cost more than bugs found earlier" the argument for code review still holds.)

Why is the effect of code review so dramatic? A lack of collaboration in the development phase may be the culprit.

With requirements and design you always have meetings. You bring in input from customers, managers, developers, and QA and synthesize a result. You do this because mistakes in requirements or architecture are expensive, possibly leading to lost sales. You debate the relative priorities, difficulty, and long-term merits of your choices.

Not so with when actually writing the source code. Individual developers type away at the tasks assigned to them. Collaboration is limited to occasional whiteboard drawings and a few shared interfaces. No one is catching the obvious bugs; no one is making sure the documentation matches the code.

Peer code review adds back the collaborative element to this phase of the software

development process.

Consider this: Nothing is commercially published without corrections from several professional editors. Find the acknowledgments in any book and you'll find reviewers who helped "remove defects." No matter how smart or diligent the author, the review process is necessary to produce a high-quality work. (And even then, what author hasn't found five more errors after seeing the first edition?)

Why do we think it's any different in software development? Why should we expect our developers to write pages of detailed code (and prose) without mistakes?

We shouldn't. If review works with novels and software design it can also work when writing code. Peer code review adds a much-needed collaborative element to the development phase of the software development process.

The $1 billion bug

In 2005, Adobe attributed $1 billion in revenue to their stronghold on the PDF format (see Adobe Systems Incorporated Letter to Stockholders FY 2005).

Why is PDF worth $1 billion? Because it's the one format that everyone can view and print (see same document). It just works. If it loses that status, Adobe loses the edifice built on that format, to which the fiscal year 2005 income statement attributes $1 billion.

Now imagine you are a development manager for Acrobat Reader, Windows Edition. The next major release is due in 9 months and you are responsible for adding five new features. You know how much is riding on Reader and how much revenue -- and jobs -- depends on its continued success.

So now the question: Which of those five features is so compelling, it would be worth introducing a crash-bug in Reader just to have that feature?

Answer: None!

Nothing is worth losing your position in the industry. But you still must implement new features to keep the technology fresh and competition at bay. You must employ every possible technique in your development process to ensure that no bugs get introduced.

Only code review will ensure that this code base -- already over ten years old -- remains maintainable for the next ten. Only code review will ensure that new hires don't make mistakes that veterans would avoid. Static code analysis and black-box QA will not do this for you. And every defect found in code review is another bug that might have gotten through QA and into the hands of a customer.

This doesn't mean they implement code review no matter what the costs; developer time is still an expensive commodity. It does mean that they're taking the time to understand this process which, if implemented properly, is a proven method for significantly reducing the number of delivered bugs, keeping code maintainable, and getting new hires productive quickly and safely.

But you don't need to have $1 billion at stake to be interested in code quality and maintainability. Delivering bugs to QA costs money; delivering bugs to customers costs a lot of money and loss of goodwill.

But if code review works this well, why don't more people talk about it? Is anyone really doing it?

Why code review is a secret

In 1991, OOP was the Next Big Thing. But strangely, at OOPSLA there were precious few papers, light on content, and yet the attendees admitted to each other in hallway talk that their companies were fervently using the new techniques and gaining significant improvements in code reusability and in breaking down complex systems.

So why weren't they talking publicly? Because the development groups that truly

understood the techniques of OOP had a competitive advantage. OOP was new and everyone was learning empirically what worked and what didn't; why give up that hard-earned knowledge to your competitors?

A successfully-implemented code review process is a competitive advantage. No one wants to give away the secret of how to release fewer defects efficiently.

When we got started no one was talking about code review in the press, so we didn't think many people were doing it. But our experience has made it clear that peer code review is widespread at companies who are serious about code quality.

But the techniques are still a secret! Peer code review has the potential to take too much time to be worth the gain in bug-fixing, code maintainability, or in mentoring new developers. The techniques that provide the benefits of peer code review while mitigating the pitfalls and managing developers' time are competitive advantages that no one wants to reveal.

Unfortunately for these successful software development organizations, we make a living making lightweight, agile code review accessible and efficient for everyone. And that's the subject of this CM Crossroads Article Series.

I'm interested. Gimme the details!

So code review works, but what if developers waste too much time doing it? What if the social ramifications of personal critiquing ruin morale? How can review be implemented in a measurable way so you can identify process problems?

In this article series we will cover case studies of review in the real world and show which conclusions you can draw from them (and which you can't). We will give our own case study of 2500 reviews. We will give pro's and con's for the five most common types of review. We will explain how to take advantage of the positive social and personal aspects of review as well as ways managers can mitigate negative emotions that can arise. We will explain how to implement a review within a CMMI/PSP/TSP context. We will give

specific advice on how to construct a peer review process that meets specific goals. Finally, we will describe a tool that our customers have used to make certain kinds of reviews as painless and efficient as possible.

Code review can be practical, efficient, and even fun. Stay tuned!

In the next article we talk about the existing formal inspection techniques and why they aren't used and developers hate them.

(Special thanks to Steven Teleki for his insights into the parallel between the rise of OOP and the modern rise of peer code review.)

