



Peer Code Review: An Agile Process

This paper was originally published by Smart Bear Software in the proceedings of the Agile Development Practices conference in November 2009.

Peer code review is one of the most effective ways to improve software quality – but is it agile? Done correctly, it absolutely is. The Agile Manifesto^[1] states:

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

Research has consistently shown that code review produces software with fewer defects, which aligns with the emphasis on working software. And what could be more interactive than two or more software developers talking (or IM'ing or emailing) about the code and making real-time improvements?

Yet many agile practitioners consider peer code review to be part of the “bad old world” of waterfall development and reject its inclusion in agile projects. This paper shows how code reviews can be conducted using methods that align perfectly with the fundamental principles of agile development.

Moving Beyond the “Code Review Stigma”

Historically, the process for conducting code review was pretty “anti-agile.” As originally conceived by Michael Fagan in 1976, *code inspections*^[4] were a heavyweight code review process that led to an entire generation of software developers who believed meetings were necessary in order to review code.

Heavyweight processes and meetings are not regarded favorably on agile projects, and that stigma has tainted the concept of code review. This “guilt by association” has worn away over time, but misconceptions still linger.

The biggest misconception is that meetings are required to do code review. Fagan stated that meetings are required, as have other researchers. But Lawrence Votta of AT&T Bell Labs was not convinced. His study^[6] showed that if developers read the code before the meeting in order to find defects, actually having a meeting will only increase the total defects found by 4% (while often tying up several hours of valuable time per participant).

Recent studies have confirmed that code review can be effective without meetings. Jason Cohen, founder of Smart Bear Software®, conducted a study^[3] at Cisco Systems® that showed that a lightweight peer code review approach was as effective as a heavyweight code inspection process, but more time-efficient by a factor of 7x.

Yet even agile devotees who recognize that meetings are not required have misconceptions about code review: “We only use the latest techniques here – code review is from the past and provides no value,” or “All the unit tests pass, so why do we need to do code reviews?” If you take away the meetings and the heavyweight process, but leave the interaction, responsiveness, and dedication to continuous improvement, then code review is very much an agile practice.

How Does Code Review Align With Agile?

Delving into the underlying principles^[2] of the Agile Manifesto provides specific evidence that code review is agile:

1. Working software is the primary measure of progress. Software developers are fallible, just like all other humans. We make mistakes. Some of those mistakes can be detected automatically (unit tests, static analysis tools, etc.) but just as professional writers have human editors in addition to spell-check software, software developers benefit from having one or more other developers examine their source code.

It is interesting how much time we spend during an iteration discussing the requirements with our stakeholder(s) and the emerging design and architecture with other software developers, but when it comes time to actually write the code the tendency is for each developer to work in isolation. The interaction during discussions of requirements, architecture, and design uncovers flaws. The same principle applies to the writing of the code.

Code reviews uncover flaws and have another key benefit that is prized by agilists – the feedback is kept close to the point of creation and happens sooner – before the code gets to QA or customers.

2. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely. That’s a tall order. To help meet it, agile teams frequently practice *collective code ownership*. The goal is that each portion of the source code is understood by more than one member of the team.

To reach that goal, it is important to pay attention to the *bus number* for each part of the code – how many team members would have to get struck by a bus before no one was left that understood the code? If the bus number for a section of the code is less than two, then that’s a problem.

By encouraging the reading and discussing of the source, code review helps maintain collective code ownership, increasing the bus number for the reviewed code. That way, if a team member is on

vacation, or leaves the team, progress can continue at the same pace.

3. Continuous attention to technical excellence and good design enhances agility. All software developers have egos and most of them are naturally curious people who enjoy learning new things.

Developers who know that their code will be reviewed tend to write better code because they know that their reputation is on the line. No one wants to be thought of as the weak link in the chain.

A corollary is that developers who review other's code get an opportunity to learn new tricks and techniques. A key step in mastering any craft is to benefit from the experience of others.

Types of Lightweight Code Review

Lightweight code review provides the right mix of code review process with agile practice, allowing effective and efficient code reviews without overwhelming burden. There are four different approaches to doing lightweight code review in an agile environment. Each has its strengths and weaknesses and they are not mutually exclusive, so there is no single right or wrong approach. As with so much in the agile world, each team needs to decide for itself which approach is correct.

1. Over the shoulder. This is the easiest technique of all: when it is time for a code review, find a developer and sit down with him or her in front of the code. Face to face communication is an easy, high-bandwidth medium for the author's explanation of the code.

An obvious drawback is that not all teams have all members in one location. An additional issue is that the reviewer is being interrupted – after the review it will take time for that developer to get back to the same level of productivity.

The biggest risk with this sort of approach, though, is that it can end up being a walk through instead of a review. If the reviewer has no prior access to the materials then typically the author does most of the talking, which can result in a passive reviewer who spends more time nodding than asking questions about the code.

2. Email pass-around. When the code is ready, send it out over email. One of the advantages of this approach is that reviewers and authors can be in different locations. Another advantage is that the reviewers can do the review at their convenience.

One obvious downside is that as the review proceeds and the emails get nested in multiple replies, it becomes more difficult to follow the conversation. And if files are reworked and line numbers change, it can be challenging to determine which version of a file – or even which line – is being referenced by a particular comment. But perhaps the biggest drawback is that it can be difficult to answer a simple

question: When is the review finished?

3. Pair programming. One of the Extreme Programming world's key contributions has been pair programming, which in some ways is a continuous code review. The advantages are that no workflow or tools or interruptions get in the way. Further, the review is at a deep level since the developer who is reviewing has the same level of experience with the code.

One obvious downside is that the time commitment is non-trivial. A less obvious downside is that the reviewer is "too close" to the code to give it a good review. After all, a key benefit of code review is to get outside opinions. If two developers are pairing to the extent that they have the exact same view of the code then the code review will likely not be as effective.

4. Tool-assisted review. Code review tools exist to help overcome the shortcomings of the approaches listed above. They can package up source files, send notifications to reviewers, facilitate communication, ensure defects are fixed, and more. The obvious downside is that they require at the very least time for installation and configuration, and in the case of commercial products, money for license purchases.

Techniques for Optimal Code Reviews

Regardless of which type of lightweight code review your team chooses, there are tips and techniques^[5] for preventing wasted time and improving the results:

1. Limit the amount of time – a developer should spend no more than sixty to ninety minutes at a time doing code review.
2. Go slowly – typically 200 to 500 lines of code per hour is the maximum rate for an effective review.
3. Limit the amount of code – as a product of the time and rate recommendations, the total amount of code for a review should be no more than 200 to 400 lines.
4. Have the author annotate the materials before the review starts. Just looking at the code using a different tool than their standard editor can lead developers to spot problems in their own code before the review begins.
5. Review checklists (if used) should be short, contain no items that are obvious or can be detected via automation, and should focus on things that are easy to forget (e.g. "Are all errors handled correctly everywhere?").

Overcoming Resistance

A key Agile Principle^[2] is: “The best architectures, requirements, and design emerge from self-organizing teams.” If peer code review is mandated by someone outside the team, its chance of success decreases. If team members do not want code review to succeed, it *will* fail.

Even when suggested by a team member, code review can still face resistance. A key to success is to start slowly. Do not attempt something like: “Starting today 100% of all code written must be peer reviewed.”

The key is to instead get the best return on time invested by initially doing code reviews only on a limited part of the source code. For example, one approach is to have the developers agree on the “top ten scariest source files” and then only review changes to those files. Or only review changes made to the stable branch of the source code. A slightly more extreme approach would be to just review unit tests – if the unit tests are complete and are passing, then those results indicate the underlying implementation is correct without investing additional review time.

The amount of code that gets reviewed can be expanded after a team has experience with code review and sees its value over time.

Summary

Peer code review is agile, when done correctly. The legacy of heavyweight code inspection processes has biased many agile developers away from code review, but there are multiple types of lightweight code review that work well in an agile environment. With the right approach and techniques, and by phasing in the use of code review over time, agile teams can more easily deliver working – and high quality – software.

About Smart Bear Software

Smart Bear is the first and most successful provider of commercial software for agile and lightweight source code reviews. Our flagship product, [Code Collaborator](#), simplifies peer review by supporting workflows, integrating with incumbent development tools, and automating audit trails, metrics, and reporting. With Code Collaborator, say goodbye to the tedious and time-wasting aspects of code reviews: no more meetings, printouts, manual metrics collection, finding a specific line number, correlating comments to code, or losing track of bugs found vs. fixed. *When code review is easy and fun, it actually gets done.*

Smart Bear is also the industry thought leader on lightweight code review and is well known for conducting the largest-ever case study of peer review at Cisco Systems. Read about the results and other insights in the book, [Best Kept Secrets of Peer Code Review](#), written by the founders and associates of Smart Bear.

For more information about lightweight code review in an agile environment, please contact Smart Bear Software. You can download a free trial of the tool at www.CodeCollaborator.com. Request a free copy of Best Kept Secrets of Peer Code Review at www.CodeReviewBook.com.

References

1. The Agile Manifesto, <http://agilemanifesto.org/>, 2001.
2. The Agile Manifesto Principles, <http://agilemanifesto.org/principles.html>, 2001.
3. Jason Cohen, *Best Kept Secrets of Peer Code Review*, <http://codereviewbook.com>, 2006.
4. M. E. Fagan. "Design and code inspections to reduce errors in program development." *IBM Systems Journal*, 15(3):216-245, 1976.
5. Smart Bear Software, *11 Best Practices of Peer Code Review*, <http://smartbear.com/docs/BestPracticesForPeerCodeReview.pdf>, 2007.
6. Lawrence G. Votta, Jr., "Does every inspection need a meeting?" *Proceedings of the 1st ACM SIGSOFT symposium on Foundations of software engineering*, p.107-114, December 08-10, 1993, Los Angeles, California, United States.