11 proven practices for more ef peer code review

Using SmartBear CodeCollaborator for lightweight code review

Jason Cohen
Published on January 25, 2011

Our team at SmartBear Software® has spent years researching existing code review from more than 6000 programmers at more than 100 companies. Clearly, people fi reviews often take too long to be practical. We used the information gleaned throug concept of lightweight code review. By using lightweight code review techniques, d time needed for full, formal code reviews. We also developed a theory for best prac efficiency and value. This article outlines those practices.

To test our conclusions about code review in general and lightweight review in parti done on code review. It encompassed 2500 code reviews, 50 programmers, and 3. For 10 months, the study tracked the MeetingPlace product team, which had memb

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

- All code had to be reviewed before it was checked into the team's vers ontr
- SmartBear's CodeCollaborator® code review software tool would be used to exp reviews.

The review process would be enforced by tools.

Metrics would be automatically collected by CodeCollaborator, which provides r

The 11 best practices, according to our study

It's common sense that peer code review (in which software developers review eac QA) identifies bugs, encourages collaboration, and keeps code more maintainable.

But it's also clear that some code review techniques are inefficient and ineffective. review process take time and kill excitement.Strict process can stifle productivity, b whether reviews are effective or even happening. And the social ramifications of pe

This article describes 11 best practices for efficient, lightweight peer code review tl scientific study and by SmartBear's extensive field experience. Use these technique your code – without wasting your developers' time. And use the latest technology t Rational Team Concert® environment.

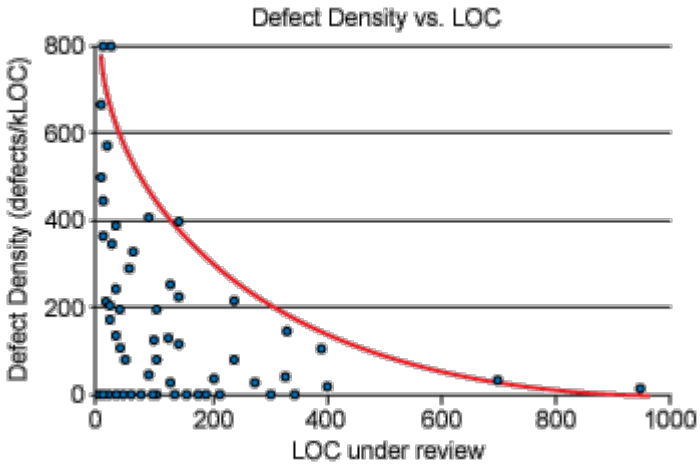
1. Review fewer than 200–400 lines of code at a time

The Cisco code review study (see the sidebar) showed that for optimal effectivenes 200-400 lines of code (LOC) at a time. Beyond that, the ability to find defects dimin over no more than 60–90 minutes, you should get a 70–90% yield. In other words, them.

The graph in Figure 1, which plots defect density against the number of lines of cod review, supports this rule. *Defect density* is the number of defects found per 1000 li code. As the number of lines of code under review grows beyond 200, defect densi considerably.

In this case, defect density is a measure of "review effectiveness." If two reviewers same code and one finds more bugs, we would consider that reviewer more effecti shows how, as we put more code in front of a reviewer, her effectiveness a ing drops. This result makes sense, because she probably doesn't have a lot of time to

Figure 1. Defect density dramatically decreases when the number of lines of inspection g
200, and it is almost zero after 400



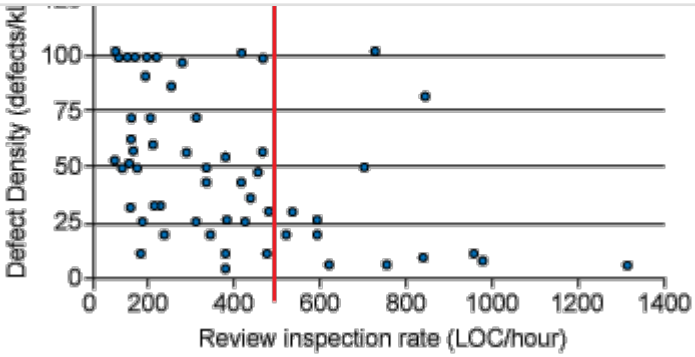
2. Aim for an inspection rate of fewer than 300–500 LOC

Take your time with code review. Faster is not better. Our research shows that you'l
optimal results at an inspection rate of less than 300–500 LOC per hour. Left to the
devices, reviewers' inspection rates will vary widely, even with similar authors, revie
files, and review sizes.

To find the optimal inspection rate, we compared *defect density* with *how fast* the re
went through the code. Again, the general result is not surprising: if you don't spenc
time on the review, you won't find many defects. If the reviewer is overwhelmed by
quantity of code, he won't give the same attention to every line as he might with a s
change. He won't be able to explore all ramifications of the change in a single sittin

So, how fast is too fast? Figure 2 shows the answer: reviewing faster than 400 LOC
results in a severe drop-off in effectiveness. And at rates above 1000 LOC per hour,
probably conclude that the reviewer isn't actually looking at the code at all.

Figure 2. Inspection effectiveness falls off when more than 500 lines of code are under r



3. Take enough time for a proper, slow review, but not m

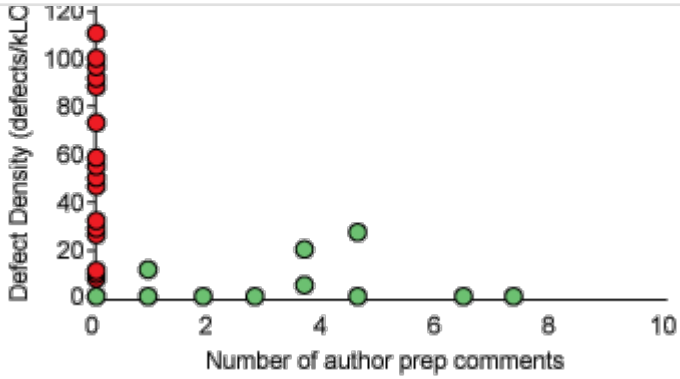
Never review code for more than 90 minutes at a stretch.
We've talked about how, for best results, you shouldn't review code too fast. But yo sitting. After about 60 minutes, reviewers simply get tired and stop finding addition supported by evidence from many other studies besides our own. In fact, it's gener activity requiring concentrated effort, performance starts dropping off after 60–90 reviewer will probably not be able to review more than 300–600 lines of code befo

On the flip side, you should always spend at least five minutes reviewing code, ever one line. Often, a single line can have consequences throughout the system, and it' five minutes to think through the possible effects that a change could have.

4. Be sure that authors annotate source code before the begins

It occurred to us that authors might be able to eliminate most defects before a revi to double-check their work, maybe reviews could be completed faster without com tell, this specific idea had not been studied before, so we tested it during the study

Figure 3. The striking effect of author preparation on defect density



The idea of *author preparation* is that authors annotate their source code before the
describe a certain behavior pattern that we measured during the study, which was
reviews. Annotations guide the reviewer through the changes, showing which files t
and methods for each code modification. These notes are not comments in the cod
reviewers.

Our theory was that, because the author has to re-think and explain the changes du
uncover many of the defects before the review even begins, thus making the review
process should yield a lower density of defects, because fewer bugs remain. Sure e
preparation had barely any defects, compared to reviews without author preparatic

We also considered a pessimistic theory to explain the lower bug findings. What if,
reviewer becomes biased or complacent so just doesn't find as many bugs? We too
investigate, and the evidence showed that the reviewers were, indeed, carefully rev
fewer bugs.

5. Establish quantifiable goals for code review, and captu improve your processes

As with any project, decide in advance on the goals of the code review process and
When you've defined specific goals, you will be able to judge whether peer review is
require.

It's best to start with external metrics, such as "reduce support calls by 20 r "h
development." This information gives you a clear picture of how your code is doing

However, it can take a while before external metrics show results. Support calls, for versions are released and in customers' hands. So it's also useful to watch internal defects are found, where your problems lie, and how long your developers are spen internal metrics for code review are *inspection rate*, *defect rate*, and *defect density*.

Consider that only automated or tightly controlled processes can give you repeatab remembering to stop and start stopwatches. For best results, use a code review toc your critical metrics for process improvement are accurate.

To improve and refine your processes, collect your metrics and tweak your process Pretty soon, you'll know exactly what works best for your team.

6. Use checklists, because they substantially improve res reviewers

Checklists are especially important for reviewers because, if the author forgets a task Checklists are a highly recommended way to check for the things that you might for authors and reviewers. Omissions are the hardest defects to find; after all, it's hard checklist is the single best way to combat the problem, because it reminds the revi something that might be missing. A checklist will remind authors and reviewers to c function arguments are tested for invalid values, and that unit tests have been crea

Another useful concept is the *personal checklist*. Each person typically makes the s your typical errors are, you can develop your own personal checklist (Personal Soft Institute, and the Capability Maturity Model Integrated recommend this practice, tc determining your common mistakes. All you have to do is keep a short checklist of t the things that you most often forget to do.

As soon as you start recording your defects in a checklist, you will start making few your mind, and your error rate will drop. We've seen this happen over and over.

Tip:

For more detailed information on checklists, plus a sample checklist, you can get a

7. Verify that the defects are actually fixed

OK, this "best practice" seems like a no-brainer. If you're going to all of the trouble
makes sense to fix them! Yet many teams that review code don't have a good way c
ensuring that bugs are actually fixed before the review is complete. It's especially c
the-shoulder reviews.

Keep in mind that these bugs aren't usually entered in Rational Team Concert logs,
released to QA. So, what's a good way to ensure that defects are fixed before the cc
using good collaborative review software, integrated with Rational Team Concert, tc
right tool, reviewers can log bugs and discuss them with the author as necessary. A
reviewers, and reviewers must then verify that each issue is resolved. The tool shou
prohibit review completion until all bugs are verified as *fixed* by the reviewer (or tra
resolved later). A work item should be approved only when the review is complete.

If you are going to go to the trouble of finding the bugs, make sure that you've fixed al
Now that you've learned these best practices for the *process* of code review, we'll d
can manage them for best results.

8. Foster a good code review culture in which finding def

Code review can do more for true team building than almost any other technique w
at a means for learning, growing, and communication. It's easy to see defects as a l
code), but fostering a negative attitude toward defects found can sour a whole tean
process.

The point of software code review is to eliminate as many defects as possible, regardl
Managers must promote the viewpoint that defects are positive. After all, each one
the goal of the bug review process is to make the code as good as possible. Every d
defect that a customer never sees and another problem that QA doesn't ha n spe

Teams need to maintain the attitude that finding defects means that the author anc

Reviews present opportunities for all developers to correct bad habits, learn new tr
Developers can learn from their mistakes, but only if they know what their issues ar
review process, the positive results disappear.

Especially if you're a junior developer or are new to a team, defects found by others
experienced peers are doing a good job in helping you become a better developer. Y
programming in a vacuum, without detailed feedback.

To maintain a consistent message that finding bugs is good, management must pro
used in performance reports. It's effective to make these kinds of promises in the o
and can call out any manager that violates a rule made so publicly.

Managers should also never use buggy code as a basis for negative performance re
sensitive to hurt feelings and negative responses to criticism and continue to remin

9. Beware of the Big Brother effect

As a developer, you automatically assume that it's true that "Big Brother is watchin
measured automatically by review-supporting tools. Did you take too long to review
many bugs in your code? How will this affect your next performance evaluation?

**Metrics should never be used to single out developers, particularly in front of their pe
morale.**
Metrics are vital for process measurement, which, in turn, provides the basis for pro
used for good or evil. If developers believe that metrics will be used against them, r
but they will probably focus on improving their metrics rather than truly writing bett

Managers can do a lot to improve the problem. First and foremost, they must be aw
that they're not propagating the impression that Big Brother is indeed scrutinizing e

Metrics should be used to measure the efficiency of the process or the effe a pr
most difficult code is handled by your most experienced developers. This code, in tu

If metrics do help a manager uncover an issue, singling someone out is likely to cau
recommend that managers deal with any issues by addressing the group as a whole
meeting for this purpose, because developers might feel uneasy if it looks like there
weekly status meeting or other normal procedure.

Managers must continue to foster the idea that finding defects is good, not bad, anc
developer's ability. Remember to make sure that it's clear to the team that defects,
introduced by a team member, shouldn't be shunned and will never be used for per

10. Review at least part of the code, even if you can't do
Ego Effect

Imagine yourself sitting in front of a compiler, tasked with fixing a small bug. But yo
finished," your peers — or worse, your boss — will be examining your work. Won't th
work, and certainly before you declare code finished, you'll be a little more conscie
immediately because you want the general timbre of the "behind your back" conve
a good developer;" not "He makes a lot of silly mistakes. When he says he's done, h

The Ego Effect drives developers to write better code because they know that othe
metrics. No one wants to be known as the guy who makes all those junior-level mis
review their own work carefully before passing it on to others.

A nice characteristic of The Ego Effect is that it works equally well whether reviews
just as "spot checks," like a random drug test. If your code has a 1 in 3 chance of be
of an incentive to make you do a great job. However, spot checks must be frequent
had just a 1 in 10 chance of getting reviewed, you might not be as diligent. You kno
make that mistake."

Reviewing 20–33% of the code will probably give you maximal Ego Effect benefit w
reviewing 20% of your code is certainly better than reviewing none.

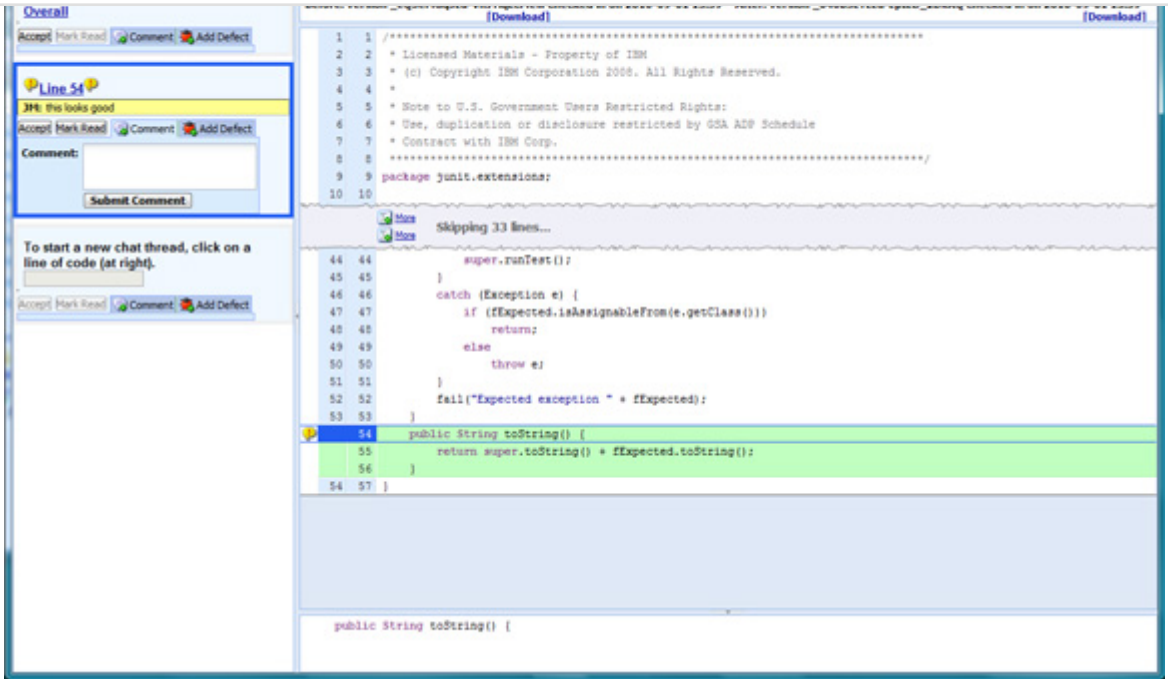
There are several main types and countless variations of code review, and these gu
However, to fully optimize the time that your team spends in review, we got optimu
review process. It's efficient, practical, and effective at finding bugs.

Formal, or *heavyweight*, inspections have been around for 30 years. They are no lor
The average heavyweight inspection takes nine hours per 200 lines of code. Althou
three to six participants and hours of painful meetings paging through code printou
organizations cannot afford to tie up people for that long, and most programmers d
recent years, many development organizations have shrugged off the yoke of meeti
and tedious metrics-gathering in favor of new *lightweight* processes that eschew fo
older, heavyweight processes.

We used our case study at Cisco to determine how the lightweight techniques comp
showed that lightweight code reviews take one-fifth the time (or less) of formal revi

Although several methods exist for lightweight code review, such as over-the-shoul
effective reviews are conducted using a collaborative software tool to facilitate the
CodeCollaborator (see Figure 4).

Figure 4. CodeCollaborator, the lightweight code review tool used in the Cisco study



CodeCollaborator is the only code review tool that integrates with IBM® Rational Te
code viewing with chat-style collaboration to free the developer from the tedium of
of code. When programmers add change sets to a work item for review, the review
CodeCollaborator with the appropriate approvers assigned. Team members can co
author and each other to work through issues, and track bugs and verify fixes. No m
scheduling required.

With a lightweight review process based on Rational Team Concert and CodeCollab
efficient reviews and fully realize the substantial benefits of code review.

So now you're armed with an arsenal of proven practices to ensure that you get the
spends on code reviews, both from a process and a social perspective. Of course, y
the benefits. Formal methods of review are simply impractical to implement for 10
would argue). Tool-assisted, lightweight code review integrated into the Rational Te
most "bang for the buck," because it offers an efficient and effective method to loca
tasks that developers hate to do. With the right tools and these practices, your tean
costly bugs before your software reaches even QA stage, so that your customers ge

For your convenience, here are the 11 practices in a simple list that's easy ep

Review fewer than 200–400 lines of code at a time.

Take enough time for a proper, slow review, but not more than 60–90 minutes.

Be sure that authors annotate source code before the review begins.

Establish quantifiable goals for code review and capture metrics so you can imp

Use checklists, because they substantially improve results for both authors and

Verify that the defects are actually fixed.

Foster a good code review culture in which finding defects is viewed positively.

Beware of the Big Brother effect.

Review at least part of the code, even if you can't do all of it, to benefit from The

Adopt lightweight, tool-assisted code reviews.

CodeCollaborator has received "Ready for IBM Rational Software" validation for Rat well as for IBM® Rational® ClearCase® and IBM® Rational® Synergy® software.

Ready for IBM Rational software

Downloadable resources

PDF of this content

Related topics

The Best Kept Secrets of Peer Code Review

CodeCollaborator

Rational Team Concert trial download

developerWorks® Learn Develop Connect

Sign in or register to add and subscribe to comments.

Subscribe me to comment notifications

developerWorks

About

Help

Submit content

Report abuse

Third-party notice

Community

Product feedback

Developer Centers

Follow us

Join

Faculty

Students

Startups

Business Partners

Select a language

English

中文

日本語

Русский

developerWorks® Learn Develop Connect

한글

Tutorials & training

Demos & sample code

Q&A forums

dW Blog

Events

Courses

Open source projects

Videos

Recipes

Downloads

APIs

Newsletters

Feeds