January 8th, 2015 by Gareth Wilson (2 min read)

Stop More Bugs with our Code Review Checklist

In our blog about effective code reviews, we recommended the use of a checklist. Checklists are a great tool in code reviews – they ensure that reviews are consistently performed throughout your team. They're also a handy way to ensure that common issues are identified and resolved.

Research by the Software Engineering Institute suggests that programmers make 15-20 common mistakes. So by adding such mistakes to a checklist, you can make sure that you spot them whenever they occur and help drive them out over time.

To get you started with a checklist, here's a list of typical items:

Code Review Checklist

General

- Does the code work? Does it perform its intended function, the logic is correct etc.
- Is all the code easily understood?
- Does it conform to your agreed coding conventions? These will usually cover location of braces, variable and function names, line length, indentations, formatting, and comments.
- Is there any redundant or duplicate code?
- Is the code as modular as possible?
- Can any global variables be replaced?
- Is there any commented out code?
- Do loops have a set length and correct termination conditions?
- Can any of the code be replaced with library functions?
- Can any logging or debugging code be removed?

Security

- Are all data inputs checked (for the correct type, length, format, and range) and encoded?
- Where third-party utilities are used, are returning errors being caught?
- Are output values checked and encoded?
- Are invalid parameter values handled?

Documentation

- Do comments exist and describe the intent of the code?
- Are all functions commented?
- Is any unusual behavior or edge-case handling described?
- Is the use and function of third-party libraries documented?

- Are data structures and units of measurement explained?
- Is there any incomplete code? If so, should it be removed or flagged with a suitable marker like 'TODO'?

Testing

- Is the code testable? i.e. don't add too many or hide dependencies, unable to initialize objects, test frameworks can use methods etc.
- Do tests exist and are they comprehensive? i.e. has at least your agreed on code coverage.
- Do unit tests actually test that the code is performing the intended functionality?
- Are arrays checked for 'out-of-bound' errors?
- Could any test code be replaced with the use of an existing API?

You'll also want to add to this checklist any language-specific issues that can cause problems.

The checklist is deliberately not exhaustive of all issues that can arise. You don't want a checklist, which is so long no-one ever uses it. It's better to just cover the common issues.

Optimize Your Checklist

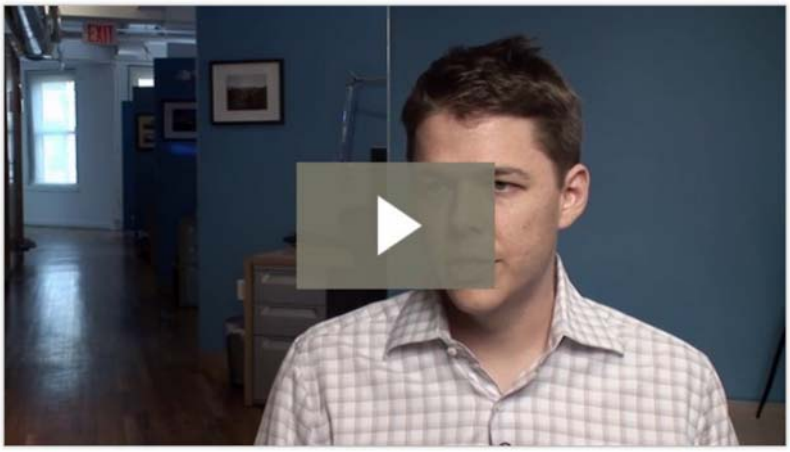
Using the checklist as a starting point, you should optimize it for your specific use-case. A great way to do this is to get your team to note the issues that arise during code reviews for a short time. With this data, you'll be able to identify your team's common mistakes, which you can then build into a custom checklist. Be sure to remove any items that don't come up (you may wish to keep rarely occurring, yet critical items such as security related issues).

Get Buy-in and Keep It Up To Date

As a general rule, any items on the checklist should be specific and, if possible, something you can make a binary decision about. This helps to avoid inconsistency in judgments. It is also a good idea to share the list with your team and get their agreement on its content. Make sure to review the checklist periodically too, to check that each item is still relevant.

Armed with a great checklist, you can raise the number of defects you detect during code reviews. This will help you to drive up coding standards and avoid inconsistent code review quality.

To learn more about code reviews at Fog Creek, check out the following video:



2

Generic Checklist for Code Reviews

Structure

- ❑ Does the code completely and correctly implement the design?
- ❑ Does the code conform to any pertinent coding standards?
- ❑ Is the code well-structured, consistent in style, and consistently formatted?
- ❑ Are there any uncalled or unneeded procedures or any unreachable code?
- ❑ Are there any leftover stubs or test routines in the code?
- ❑ Can any code be replaced by calls to external reusable components or library functions?
- ❑ Are there any blocks of repeated code that could be condensed into a single procedure?
- ❑ Is storage use efficient?
- ❑ Are symbolics used rather than "magic number" constants or string constants?
- ❑ Are any modules excessively complex and should be restructured or split into multiple routines?

Documentation

- ❑ Is the code clearly and adequately documented with an easy-to-maintain commenting style?
- ❑ Are all comments consistent with the code?

Variables

- ❑ Are all variables properly defined with meaningful, consistent, and clear names?
- ❑ Do all assigned variables have proper type consistency or casting?
- ❑ Are there any redundant or unused variables?

Arithmetic Operations

- ❑ Does the code avoid comparing floating-point numbers for equality?
- ❑ Does the code systematically prevent rounding errors?
- ❑ Does the code avoid additions and subtractions on numbers with greatly different magnitudes?
- ❑ Are divisors tested for zero or noise?

Loops and Branches

- ❑ Are all loops, branches, and logic constructs complete, correct, and properly nested?
- ❑ Are the most common cases tested first in IF- -ELSEIF chains?
- ❑ Are all cases covered in an IF- -ELSEIF or CASE block, including ELSE or DEFAULT clauses?
- ❑ Does every case statement have a default?
- ❑ Are loop termination conditions obvious and invariably achievable?
- ❑ Are indexes or subscripts properly initialized, just prior to the loop?
- ❑ Can any statements that are enclosed within loops be placed outside the loops?
- ❑ Does the code in the loop avoid manipulating the index variable or using it upon exit from the loop?

Defensive Programming

- ❑ Are indexes, pointers, and subscripts tested against array, record, or file bounds?
- ❑ Are imported data and input arguments tested for validity and completeness?
- ❑ Are all output variables assigned?
- ❑ Are the correct data operated on in each statement?
- ❑ Is every memory allocation deallocated?
- ❑ Are timeouts or error traps used for external device accesses?
- ❑ Are files checked for existence before attempting to access them?
- ❑ Are all files and devices are left in the correct state upon program termination?

Code Review Checklist

http://commondatastorage.googleapis.com/bluelotussoftware/documents/Code%20Review%20Checklist.docx

Documentation

- ☐ All methods are commented in clear language. If it is unclear to the reader, it is unclear to the user.
- ☐ All source code contains @author for all authors.
- ☐ @version should be included as required.
- ☐ All class, variable, and method modifiers should be examined for correctness.
- ☐ Describe behavior for known input corner-cases.
- ☐ Complex algorithms should be explained with references. For example, document the reference that identifies the equation, formula, or pattern. In all cases, examine the algorithm and determine if it can be simplified.
- ☐ Code that depends on non-obvious behavior in external frameworks is documented with reference to external documentation.
- ☐ Confirm that the code does not depend on a bug in an external framework which may be fixed later, and result in an error condition. If you find a bug in an external library, open an issue, and document it in the code as necessary.
- ☐ Units of measurement are documented for numeric values.
- ☐ Incomplete code is marked with //TODO or //FIXME markers.
- ☐ All public and private APIs are examined for updates.

Testing

- ☐ Unit tests are added for each code path, and behavior. This can be facilitated by tools like Sonar, and Cobertura.
- ☐ Unit tests **must** cover error conditions and invalid parameter cases.
- ☐ Unit tests for standard algorithms should be examined against the standard for expected results.
- ☐ Check for possible null pointers are always checked before use.
- ☐ Array indices are always checked to avoid `ArrayIndexOfBounds` exceptions.
- ☐ Do not write a new algorithm for code that is already implemented in an existing public framework API, and tested.
- ☐ Ensure that the code fixes the issue, or implements the requirement, and that the unit test confirms it. If the unit test confirms a fix for issue, add the issue number to the documentation.

Error Handling

- ☐ Invalid parameter values are handled properly early in methods (Fast Fail).

- ☐ `NullPointerException` conditions from method invocations are checked.
- ☐ Consider using a general error handler to handle known error conditions.
- ☐ An Error handler **must** clean up state and resources no matter where an error occurs.
- ☐ Avoid using `RuntimeException`, or sub-classes to avoid making code changes to implement correct error handling.
- ☐ Define and create custom Exception sub-classes to match your specific exception conditions. Document the exception in detail with example conditions so the developer understands the conditions for the exception.
- ☐ (JDK 7+) Use try-with-resources. (JDK < 7) check to make sure resources are closed.
- ☐ **Don't pass the buck!** Don't create classes which throw Exception rather than dealing with exception condition.
- ☐ **Don't swallow exceptions!** For example `catch (Exception ignored) {}`. It should at least log the exception.

Thread Safety

- ☐ Global (static) variables are protected by locks, or locking sub-routines.
- ☐ Objects accessed by multiple threads are accessed only through a lock, or synchronized methods.
- ☐ Locks must be acquired and released in the right order to prevent deadlocks, even in error handling code.

Performance

- ☐ Objects are duplicated only when necessary. If you must duplicate objects, consider implementing Clone and decide if deep cloning is necessary.
- ☐ No busy-wait loops instead of proper thread synchronization methods. For example, avoid `while(true){ ... sleep(10);...}`
- ☐ Avoid large objects in memory, or using String to hold large documents which should be handled with better tools. For example, don't read a large XML document into a String, or DOM.
- ☐ Do not leave debugging code in production code.
- ☐ Avoid `System.out.println();` statements in code, or wrap them in a Boolean condition statement like `if(DEBUG) {...}`
- ☐ "Optimization that makes code harder to read should only be implemented if a profiler or other tool has indicated that the routine stands to gain from optimization. These kinds of optimizations should be well documented and code that performs the same task should be preserved." — **UNKNOWN**.

Code Review Checklist

Coding Standards
- understandable
- adhere code guidelines
- indentation
- no magic numbers
- naming
- units, bounds
- spacing: horizontal (btwn operators, keywords) and vertical (btwn methods, blocks)

Comments
- no needless comments
- no obsolete comments
- no redundant comments
- methods document parameters it modifies, functional dependencies
- comments consistent in format, length, level of detail
- no code commented out

Logic
- array indexes within bounds
- conditions correct in ifs, loops
- loops always terminate
- division by zero
- refactor statements in the loop to outside the loop

Error Handling
- error messages understandable and complete
- edge cases (null, 0, negative)
- parameters valid
- files, other input data valid

Code Decisions
- code at right level of abstraction
- methods have appropriate number, types of parameters
- no unnecessary features
- redundancy minimized
- mutability minimized
- static preferred over nonstatic
- appropriate accessibility (public, private, etc.)
- enums, not int constants
- defensive copies when needed
- no unnecessary new objects
- variables in lowest scope
- objects referred to by their interfaces, most generic supertype

Review Information:

Name of Reviewer: _____

Name of Coder: _____

File(s) under review: _____

Brief description of change being reviewed: _____

_____

Review Notes (problems or decisions):

SVN Versions (if applicable):
Before review: _____
After revisions: _____

SMARTBEAR

Ensuring Software Success℠

11 Best Practices for Peer Code Review

A SmartBear White Paper

Using peer code review best practices optimizes your code reviews, improves your code and makes the most of your developers' time. The recommended best practices contained within for efficient, lightweight peer code review have been proven to be effective via extensive field experience.

Contents

Introduction

It's common sense that peer code review – in which software developers review each other's code before releasing software to QA – identifies bugs, encourages collaboration, and keeps code more maintainable.

But it's also clear that some code review techniques are inefficient and ineffective. The meetings often mandated by the review process take time and kill excitement. Strict process can stifle productivity, but lax process means no one knows whether reviews are effective or even happening. And the social ramifications of personal critique can ruin morale.

This whitepaper describes 11 best practices for efficient, lightweight peer code review that have been proven to be effective by scientific study and by SmartBear's extensive field experience. Use these techniques to ensure your code reviews improve your code – without wasting your developers' time.

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

The Cisco code review study (see sidebar on page 5) shows that for optimal effectiveness, developers should review fewer than 200-400 lines of code (LOC) at a time. Beyond that, the 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.

The graph to the right, which plots defect density against the number of lines of code under review, supports this rule. Defect density is the number of defects per 1000 lines of code. As the number of lines of code under review grows beyond 300, defect density drops off considerably.

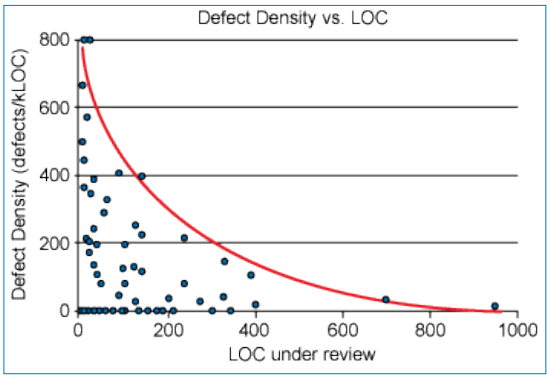


Figure 1: Defect density dramatically decreases when the number of lines of inspection goes above 200, and is almost zero after 400.

In this case, defect density is a measure of "review effectiveness." If two reviewers review the same code and one finds more bugs, we would consider her more effective. Figure 1 shows how, as we put more code in front of a reviewer, her effectiveness at finding defects drops. This result makes sense – the reviewer probably doesn't have a lot of time to spend on the review, so inevitably she won't do as good a job on each file.

2. Aim for an inspection rate of less than 300-500 LOC/hour.

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

To find the optimal inspection rate, we compared 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. If the reviewer is overwhelmed by a large quantity of code, he won't give the same attention to

every line as he might with a small change. He won't be able to explore all ramifications of the change in a single sitting.

So – how fast is too fast? Figure 2 shows the answer: 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.

Important Definitions

- ***Inspection Rate:*** *How fast are we able to review code? Normally measured in kLOC (thousand Lines Of Code) per man-hour.*
- ***Defect Rate:*** *How fast are we able to find defects? Normally measured in number of defects found per man-hour.*
- ***Defect Density:*** *How many defects do we find in a given amount of code (not how many there are)? Normally measured in number of defects found per kLOC.*

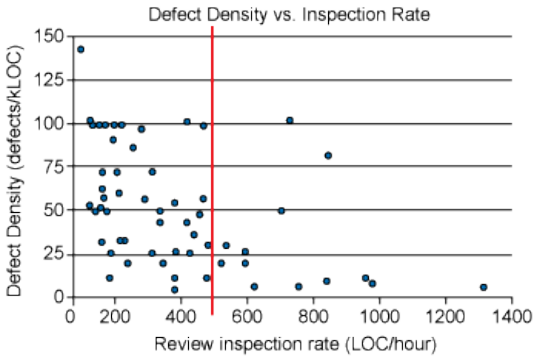


Figure 2: Inspection effectiveness falls of when greater than

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

We've talked about how you shouldn't review code too fast for best results – but you also shouldn't review too long in one sitting. After about 60 minutes, reviewers simply wear out and stop finding additional defects. This conclusion is well-effectiveness. And at rates above 1000 LOC/hour, you can probably conclude that the reviewer isn't actually looking at the code at all. 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.

Given these human limitations, a reviewer will probably not be able to review more than 300-600 lines of code before his performance drops. 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.

You should never review code for more than 90 minutes at a stretch.

4. Authors should annotate source code before the review begins.

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 idea of "author preparation" is that authors should annotate their source code before the review begins. We

invented the term to describe a certain behavior pattern we measured during the study, exhibited by about 15 % of the reviews. 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.

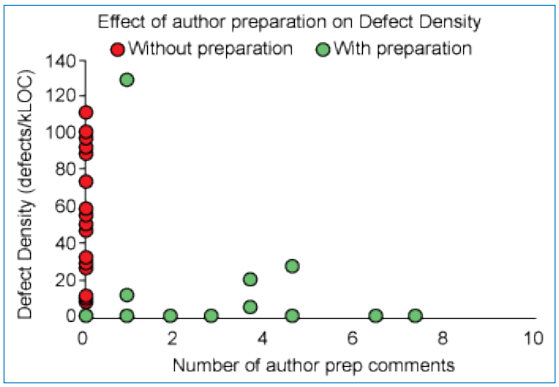


Figure 3: The striking effect of author preparation on defect density.

Our theory was that because the author has to re-think and explain the changes during the annotation process, the author will himself himself uncover many of the defects before the review even begins, thus making the review itself more efficient. As such, the review process should yield a lower defect density, since fewer bugs remain.

We also considered a pessimistic theory to explain the lower bug findings. What if, when the author makes a comment, the reviewer becomes biased or complacent, and just doesn't find as many bugs? We took a random sample of 300 reviews to investigate, and the evidence definitively showed that the reviewers were indeed carefully reviewing the code – there were just fewer bugs.

> Sure enough, reviews with author preparation have barely any defects compared to reviews without author preparation.

5. Establish quantifiable goals for code review and capture metrics so you can improve your processes.

As with any project, you should decide in advance on the goals of the code review process and how you will measure its effectiveness. Once you've defined specific goals, you will be able to judge whether peer review is really achieving the results you require.

It's best to start with external metrics, such as "reduce support calls by 20 %," or "halve the percentage of defects injected by development." This information gives you a clear picture of how your code is doing from the outside perspective, and it should have a quantifiable measure – not just a vague "fix more bugs."

However, it can take a while before external metrics show results. Support calls, for example, won't be affected until new versions are released and in customers' hands. So it's also useful to watch internal process metrics to get an idea of how many defects are found, where your problems lie, and how long your developers are spending on reviews. The most common internal metrics for code review are *inspection rate, defect rate, and defect density*.

Consider that only automated or tightly-controlled processes can give you repeatable metrics – humans aren't good at remembering to stop and start stopwatches. For best results, use a code review tool that gathers metrics *automatically* so that your critical metrics for process improvement are accurate.

To improve and refine your processes, collect your metrics and tweak your processes to see how changes affect

your results. Pretty soon you'll know exactly what works best for your team.

6. Checklists substantially improve results for both authors and reviewers.

Checklists are a highly recommended way to find the things you forget to do, and are useful for both authors and reviewers. 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 will remind 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.

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

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 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. We've seen this happen over and over.

For more detailed information on checklists plus a sample checklist, get yourself a free copy of the book, Best Kept Secrets of Peer Code Review.

7. Verify that defects are actually fixed!

OK, this "best practice" seems like a no-brainer. If you're

The World's Largest Code Review Study at Cisco Systems®

Our team at SmartBear Software® has spent years researching existing code review studies and collecting "lessons learned" from more than 6000 programmers at 200+ companies. Clearly people find bugs when they review code – but the reviews often take too long to be practical! We used the information gleaned through years of experience to create the concept of lightweight code review. Using lightweight code review techniques, developers can review code in 1/5th the time needed for full "formal" code reviews. We also developed a theory for best practices to employ for optimal review efficiency and value, which are outlined in this white paper.

To test our conclusions about code review in general and lightweight review in particular, we conducted the world's largest-ever published study on code review, encompassing 2500 code reviews, 50 programmers, and 3.2 million lines of code at Cisco Systems. For ten months, the study tracked the MeetingPlace® product team, distributed across Bangalore, Budapest, and San José.

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.
- SmartBear's CodeCollaborator® 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 CodeCollaborator, which provides review-level and summary-level reporting.

going to all of the trouble of reviewing code to find bugs, it certainly makes sense to fix them! Yet many teams who review code don't have a good way of tracking defects found during review, and ensuring that bugs are actually fixed before the review is complete. It's especially difficult to verify results in e-mail or over-the-shoulder reviews.

More on the Cisco Study...

After ten months of monitoring, the study crystallized our theory: done properly, lightweight code reviews are just as effective as formal ones – but are substantially faster (and less annoying) to conduct! Our lightweight reviews took an average of 6.5 hours less time to conduct than formal reviews, but found just as many bugs.

Besides confirming some theories, the study uncovered some new rules, many of which are outlined in this paper. Read on to see how these findings can help your team produce better code every day.

Keep in mind that these bugs aren't usually logged in the team's usual defect tracking system, because they are bugs found before code is released to QA, often before it's even checked into version control. So, what's a good way to ensure that defects are fixed before the code is given the All Clear sign? We suggest using good collaborative review software to track defects found in review. With the right tool, reviewers can logs bugs and discuss them with the author. Authors then fix the problems and notify reviewers, and reviewers must verify that the issue is resolved. The tool should track bugs found during review and prohibit review completion until all bugs are verified fixed by the reviewer (or consciously postponed to future releases and tracked using an established process).

The point of software code review is to eliminate as many defects as possible – regardless of who "caused" the error.

If you're going to go to the trouble of finding the bugs, make sure you've fixed them all!

Now that you've learned best practices for the process of code review, we'll discuss some social effects and how you can manage them for best results.

8. Managers must foster a good code review culture in which finding defects is viewed positively.

Code review can do more for true team building than almost any other technique we've seen – but only if managers promote it at a means for learning, growing, and communication. It's easy to see defects as a bad thing – after all they are mistakes in the code – but fostering a negative attitude towards defects found can sour a whole team, not to mention sabotage the bug-finding process.

Managers must promote the viewpoint that defects are positive. After all, each one is an opportunity to improve the code, and the goal of the bug review process is to make the code as good as possible. Every defect found and fixed in peer review is a defect a customer never saw, another problem QA didn't have to spend time tracking down.

Teams should maintain the attitude that finding defects means the author and reviewer have successfully worked as a team to jointly improve the product. It's not a case of "the author made a defect and the review found it." It's more like a very efficient form of pair-programming.

Reviews present opportunities for all developers to correct bad habits, learn new tricks and expand their capabili-

ties. Developers can learn from their mistakes – but only if they know what their issues are. And if developers are afraid of the review process, the positive results disappear.

Especially if you're a junior developer or are new to a team, defects found by others are a good sign that your more experienced peers are doing a good job in helping you become a better developer. You'll progress far faster than if you were programming in a vacuum without detailed feedback.

To maintain a consistent message that finding bugs is good, management must promise that defect densities will never be used in performance reports. It's effective to make these kinds of promises in the open – then developers know what to expect and can call out any manager that violates a rule made so public.

Managers should also never use buggy code as a basis for negative performance review. They must tread carefully and be sensitive to hurt feelings and negative responses to criticism, and continue to remind the team that finding defects is good.

9. Beware the "Big Brother" effect.

"Big Brother is watching you." As a developer, you automatically assume it's true, especially if your review metrics are measured automatically by review-supporting tools. Did you take too long to review some changes? Are your peers finding too many bugs in your code? How will this affect your next performance evaluation?

Metrics are vital for process measurement, which in turn provides the basis for process improvement. But metrics can be used for good or evil. If developers believe that metrics will be used against them, not only will they be hostile to the process, but they will probably focus on improving their metrics rather than truly writing better code and being more productive.

Metrics should never be used to single out developers, particularly in front of their peers. This practice can seriously damage morale.

Managers can do a lot to improve the problem. First and foremost – they should be aware of it and keep an eye out to make sure they're not propagating the impression that Big Brother is indeed scrutinizing every move.

Metrics should be used to measure the efficiency of the process or the effect of a process change. Remember that often the most difficult code is handled by your most experienced developers; this code in turn is more likely to be more prone to error – as well as reviewed heavily (and thus have more defects found). So large numbers of defects are often more attributable to the complexity and risk of a piece of code than to the author's abilities.

If metrics do help a manager uncover an issue, singling someone out is likely to cause more problems than it solves. We recommend that managers instead deal with any issues by addressing the group as a whole. It's best not to call a special meeting for this purpose, or developers may feel uneasy because it looks like there's a problem. Instead, they should just roll it into a weekly status meeting or other normal procedure.

Managers must continue to foster the idea that finding defects is good, not evil, and that defect density is not correlated with developer ability. Remember to make sure it's clear to the team that defects, particularly the number of defects introduced by a team member, shouldn't be shunned and will never be used for performance evaluations.

10. The Ego Effect: Do at least some code review, even if you don't have time to review it all.

Imagine yourself sitting in front of a compiler, tasked with fixing a small bug. But you know that as soon as you say "I'm finished," your peers – or worse, your boss – will be critically examining your work. Won't this change your development style? As you work, and certainly before you declare code-complete, you'll be a little more conscientious. You'll be a better developer immediately because you want the general timbre of the "behind your back" conversations to be, "His stuff is pretty tight. He's a good developer;" not "He makes a lot of silly mistakes. When he says he's done, he's not."

The "Ego Effect" drives developers to write better code because they know that others will be looking at their code and their metrics. And no one wants to be known as the guy who makes all those junior-level mistakes. The Ego Effect drives developers to 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 are mandatory for all code changes or just used as "spot checks" like a random drug test. If your code has a 1 in 3 chance of being called out for review, that's still enough of an incentive to make you do a great job. However, spot checks must be frequent enough to maintain the Ego Effect. If you had just a 1 in 10 chance of getting reviewed, you might not be as diligent. You know you can always say, "Yeah, I don't usually do that."

Reviewing 20-33 % of the code will probably give you maximal Ego Effect benefit with minimal time expenditure, and reviewing 20 % of your code is certainly better than none!

11. Lightweight-style code reviews are efficient, practical, and effective at finding bugs.

There are several main types, and countless variations, of code review, and the best practices you've just learned will work with any of them. However, to fully optimize the time your team spends in review, we recommend a tool-assisted lightweight review process.

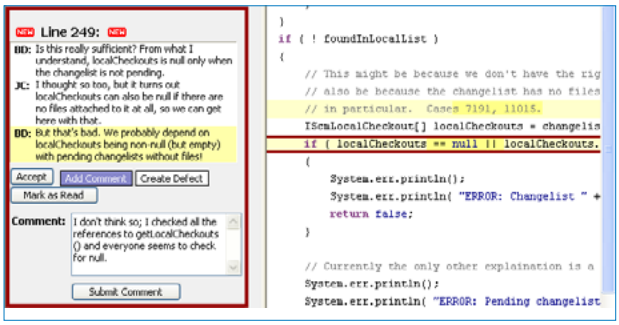


Figure 4: CodeCollaborator, the lightweight code review tool used in the Cisco study.

Formal, or heavyweight, inspections have been around for 30 years – and they are no longer the most efficient way to review code. The average heavyweight inspection takes nine hours per 200 lines of code. While effective, this rigid process requires three to six participants and hours of painful meetings paging through code print-outs in exquisite detail. Unfortunately, most organizations can't afford to tie up people for that long – and most programmers despise the tedious process required. In recent years, many development organizations have shrugged off the yoke of meeting schedules, paper-based code readings, and tedious metrics-gathering in favor of new lightweight processes that eschew formal meetings and lack the overhead of the older, heavy-weight processes.

We used our case Study at Cisco to determine how the lightweight techniques compare to the formal processes.

The results showed that lightweight code reviews take 1/5th the time (or less!) of formal reviews and they find just as many bugs!

While several methods exist for lightweight code review, such as "over the shoulder" reviews and reviews by email, the most effective reviews are conducted using a collaborative software tool to facilitate the review. A good lightweight code review tool integrates source code viewing with "chat room" collaboration to free the developer from the tedium of associating comments with individual lines of code. These tools package the code for the author, typically with version control integration, and then let other developers comment directly on the code, chat with the author and each other to work through issues, and track bugs and verify fixes. No meetings, print-outs, stop-watches, or scheduling required. With a lightweight review process and a good tool to facilitate it, your team can conduct the most efficient reviews possible and can fully realize the substantial benefits of code review.

Summary

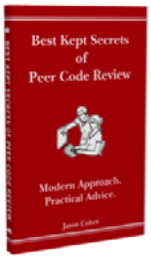


Figure 5: Best Kept Secrets of Peer Code Review – the only book to address lightweight code review.

So now you're armed with an arsenal of best practices to ensure that you get the most of out your time spent in code reviews – both from a process and a social perspective. Of course you have to actually do code reviews to realize the benefits. Old, formal methods of review are simply impractical to implement for 100 % of your code (or any percent, as some would argue). Tool-assisted lightweight code review provides the most "bang for the buck," offering both an efficient and effective method to locate defects – without requiring painstaking tasks that developers hate to do. With the right tools and best-practices, your team can peer-review all of its code, and find costly bugs before your software reaches even QA – so your customers get top-quality products every time!

More details on these best practices, the case study, and other topics are chronicled in Jason Cohen's book, Best Kept Secrets for Peer Code Review, currently available FREE.

For information on SmartBear Software's CodeCollaborator code review tool, please contact us!

Download now to try CodeCollaborator for free!

You may also enjoy these other resources in the SmartBear Software Quality Series:

- Uniting Your Automated and Manual Test Efforts
- 6 Tips to Get Started with Automated Testing

Be smart and join our growing community of over 100,000 development, QA and IT professionals in 90 countries.

Try CodeCollaborator Free for 30 Days

Scan and download your free trial to see why users choose SmartBear for peer code review.

About SmartBear Software

SmartBear Software provides tools for over one million software professionals to build, test, and monitor some of the best software applications and websites anywhere – on the desktop, mobile and in the cloud. Our users can be found worldwide, in small businesses, Fortune 100 companies, and government agencies. Learn more about the SmartBear Quality Anywhere Platform, our award-winning tools, or join our active user community at www.smartbear.com, on Facebook, or follow us on Twitter @smartbear.

SmartBear Software, Inc. 100 Cummings Center, Suite 234N Beverly, MA 01915
+1 978.236.7900 www.smartbear.com CC-WP-11BP-42012-WEB