Methods & Tools Software Development Magazine - Programming, Software Testing, Project Management, Jobs

Automated Testing of Desktop, Web and Mobile with Ranorex

Agile Development East Conference

SpiraTeam Agile Project Tool

Enov8 Agile Tools

Online Training

Four Ways to a Practical Code Review

Jason Cohen, Smart Bear Software, http://www.smartbearsoftware.com/

How to almost get kicked out of a meeting

Two years ago I was *not invited* to a meeting with the CTO of a billion-dollar software development shop, but I didn't know that until I walked in the room. I had been asked by the head of Software Process and Metrics to come and talk about a new type of lightweight code review that we had some successes with.

But the CTO made it clear that my presence was Not Appreciated.

"You see," he explained, "we already do code inspections. Michael Fagan invented inspections in 1976 and his company is teaching us how to do it." His face completed the silent conclusion: "And you sir, are no Michael Fagan."

"Currently 1% of our code is inspected," offered the process/metrics advocate. "We believe by the end of the year we can get it up to 7%." Here Mr. Metrics stopped and shot a glance over to Mr. CTO. The latter's face fell. Whatever was coming, they obviously had had this discussion before.

"The problem is we can't inspect more than that. Given the number of hours it takes to complete a Fagan inspection, we don't have the time to inspect more than 7% of the new code we write."

My next question was obvious: "What are you going to do about the other 93%?" Their stares were equally obvious – my role here was to convince the CTO that we had the answer.

This story has a happy ending, but before we get there I have to explain what it means to "inspect" code because this is what most developers, managers, and process engineers think of when they hear "code review." It's the reason this company couldn't review 93% of their code and why developers *hate* the idea. And changing this notion of what it means to "review code" means liberating developers so they can get the benefits of code review without the heavy-weight process of a formal inspection.

Michael Fagan – father of a legacy

If you've ever read anything on peer code review you know that Michael Fagan is credited with the first published, formalized system of code review. His technique, developed at IBM in the mid-1970's, demonstrably removed defects from any kind of document from design specs to OS/370 assembly code. To this day, any technique resembling his carries his moniker of "code inspection."

Take a deep breath...

I'm going to describe a "code inspection" in brief, but brace yourself. This is heavyweight process at its finest, so bear with me. It will all be over soon, I promise. A code inspection consists of seven phases. In the Planning Phase the author gathers Materials, ensures that they meet the pre-defined Entry Criteria, and determines who will participate in the inspection. There are four participants with four distinct roles: The Author, the Moderator, the Reviewer, and the Reader. Sometimes there is an Observer. All participants need to be invited to the first of several meetings, and this meeting must be scheduled with the various participants. This first meeting kicks off the Introduction Phase where the Author explains the background, motivation, and goals for the review. All participants get printed copies of the Materials. (This important — it's not a

Fagan Inspection unless it's printed out.) The participants schedule the next meeting and leave. This starts the Reading Phase where each person reads the Materials, but each role reads for a different purpose and — this is very important — no one identifies defects. When the next meeting convenes this starts the Inspection Phase. The Moderator sets the pace of this meeting and makes sure everyone is performing their role and not ruining anything with personal attacks. The Reader presents the Materials because it was his job to "read for comprehension" since often someone else's misunderstanding indicates a fault in the Materials. During the meeting a Defect Log is kept so the Author will know what needs to be fixed. Before the meeting ends, they complete a rubric that will help with later process improvement. If defects were found the inspection enters the Rework Phase where the Author fixes the problems, and later there will be a Verification Phase to make sure the fixes were appropriate and didn't open new defects. Finally the inspection can enter the Completed Phase.

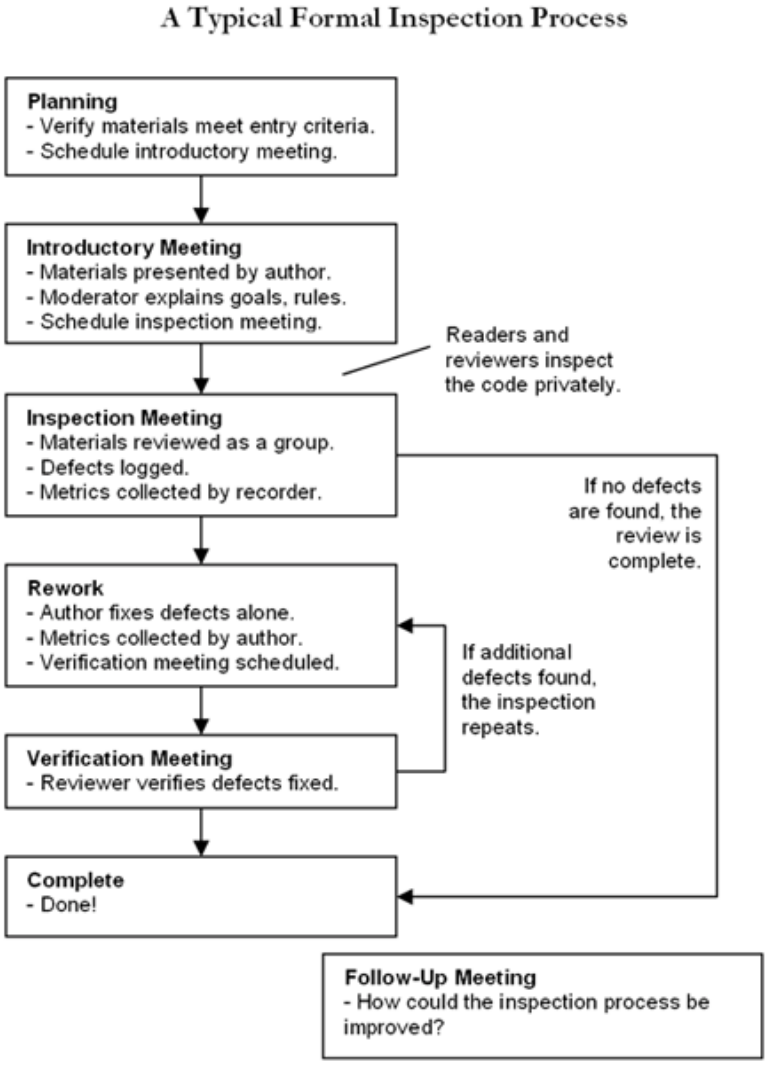

Figure 1: Typical workflow for a "formal" inspection.

Not shown are the artifacts created by the review: The defect log, meeting notes, and metrics log. Some inspections also have a closing questionnaire used in the follow-up meeting

...you can let it out now

The good news is, this works. It uncovers defects, it helps when training new hires, and the whole process can be measured for process insight and improvement. If you have extra money laying around in your budget, Mr. Fagan himself will even come show you how to do it.

The bad news should be obvious in this day of Agile Methodologies. Studies show that the average inspection takes 9 man-hours per 200 lines of code, so of *course* Mr. CTO couldn't do this for every code change in the company.

Over the years there have been experiments, case studies, and books on this subject, almost always using some form of "code inspection" as the basis. In our survey of published case studies and experiments in the past 20 years, we found that 95% of them tried inspections only in small pilot groups, and that in *no case* were they able to apply the technique to all their software development projects.

If "Agile" can do it, why can't we?

But surely there is another way. Fagan inspections were designed in the days when business logic was written in assembly language and "computer science" wasn't a major and dinosaurs roamed the earth.

Have we learned nothing since then? Don't we need different techniques when reading object-oriented code in a 3-tier application? Don't the challenges of off-shore development require new processes? Hasn't the rise of Agile Methodologies shown us that we can have process and metrics and measurement and improvement and happy developers all at the same time?

So finish the story already!

By now you can guess how the story ends. Using arguments not unlike those above, Mr. Metrics and I convinced Mr. CTO to at least try our lightweight code review technique in a pilot program with a one development group that was already hopelessly opposed to Fagan inspections. The metrics that came out of that group demonstrated the effectiveness of the lightweight system, and within 18 months Code Collaborator was deployed across the entire organization.

What does "lightweight" mean?

Assuming you've bought into the argument that code review is good but heavyweight inspection process is not practical, the next question is: How do we make reviews practical?

We'll explore four lightweight techniques:

1. **Over-the-shoulder:** One developer looks over the author's shoulder as the latter walks through the code.
2. **Email pass-around:** The author (or SCM system) emails code to reviewers
3. **Pair Programming**: Two authors develop code together at the same workstation.
4. **Tool-assisted:** Authors and reviewers use specialized tools designed for peer code review.

Over-the-shoulder reviews

This is the most common and informal (and easiest!) of code review. An "over-the-shoulder" review is just that — a developer standing over the author's workstation while the author walks the reviewer through a set of code changes.

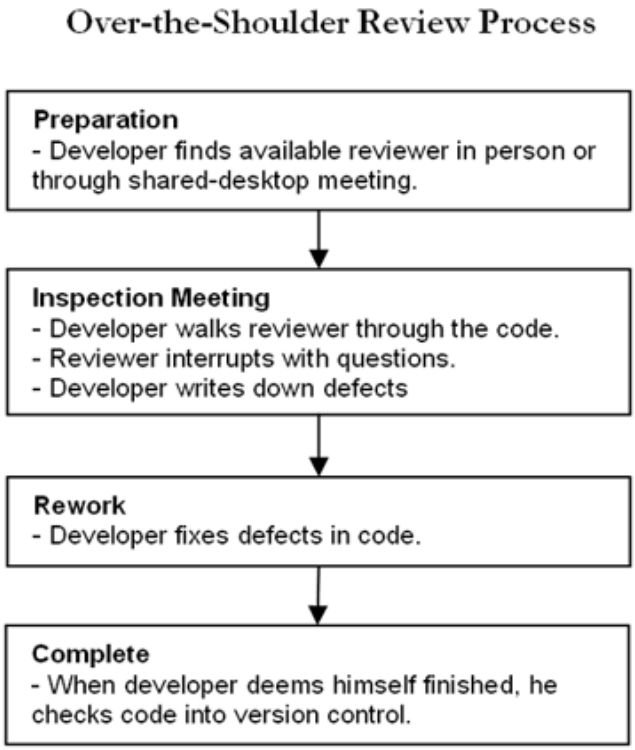

Figure 1: A typical Over-the-shoulder code walk-through process. Typically, no review artifacts are created.

Typically, the author "drives" the review by sitting at the keyboard and mouse, opening various files, pointing out the changes and explaining what he did. The author can present the changes using various tools and even go back and forth between changes and other files in the project. If the reviewer sees something amiss, they can engage in a little "spot pair-programming" as the author writes the fix while the reviewer hovers. Bigger changes where the reviewer doesn't need to be involved are taken off-line.

With modern desktop-sharing software a so-called "over-the-shoulder" review can be made to work over long distances, although this can complicate the process because you need to schedule these sharing meetings and communicate over the phone.

The most obvious advantage of over-the-shoulder reviews is simplicity in execution. Anyone can do it, any time, without training. It can also be deployed whenever you need it most — an especially complicated change or an alteration to a "stable" code branch.

Before I list out the pros and cons, I'd like you to consider a certain effect that only this type of review exhibits. Because the author is controlling the pace of the review, often the reviewer doesn't get a chance to do a good job. The reviewer might not be given enough time to ponder a complex portion of code. The reviewer doesn't get a chance to poke around other source files to check for side-effects or verify that API's are being used correctly.

The author might explain something that clarifies the code to the reviewer, but the next developer who reads that code won't have the advantage of that explanation unless it is encoded as a comment in the code. It's difficult for a reviewer to be objective and aware of these issues while being driven through the code with an expectant developer peering up at him.

So:

- Pro: Easy to implement
- Pro: Fast to complete
- Pro: Might work remotely with desktop-sharing and conference calls
- Con: Reviewer led through code at author's pace
- Con: Usually no verification that defects are really fixed
- Con: Easy to accidentally skip over a changed file
- Con: Impossible to enforce the process

- Con: No metrics or process measurement/improvement

Email pass-around reviews

This is the second-most common form of lightweight code review, and the technique preferred by most open-source projects. Here, whole files or changes are packaged up by the author and sent to reviewers via email. Reviewers examine the files, ask questions and discuss with the author and other developers, and suggest changes.

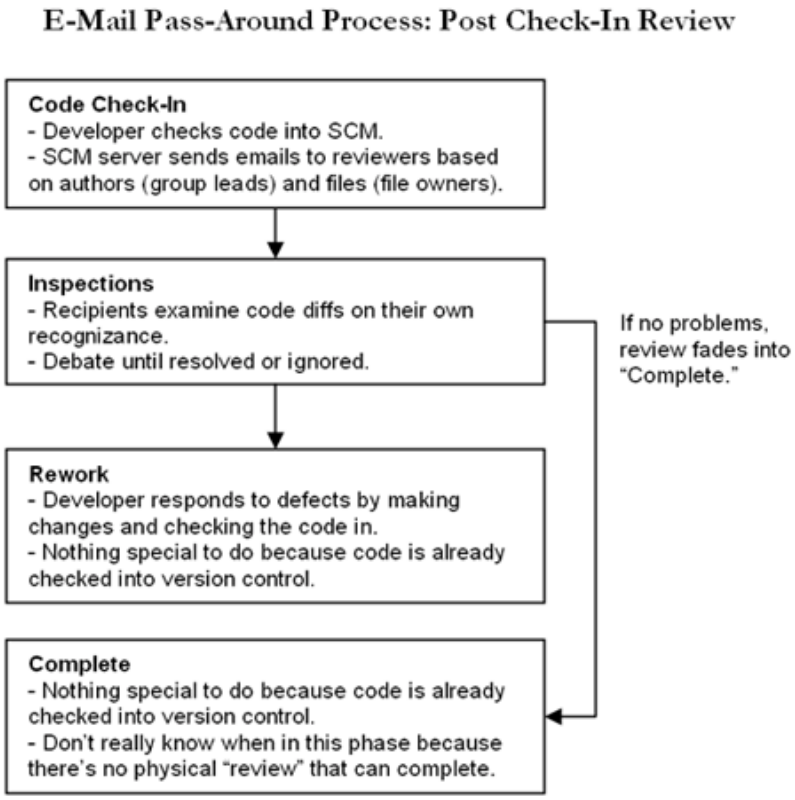

Figure 3. Typical process for an e-mail pass-around review for code already checked into a version control system. These phases are not this distinct in reality because there's no tangible "review" object.

The hardest part of the email pass-around is in finding and collecting the files under review. On the author's end, he has to figure out how to gather the files together. For example, if this is a review of changes being proposed to check into version control, the user has to identify all the files added, deleted, and modified, copy them somewhere, then download the previous versions of those files (so reviewers can see what was changed), and organize the files so the reviewers know which files should be compared with which others. On the reviewing end, reviewers have to extract those files from the email and generate differences between each.

The version control system can assist the process by sending the emails out automatically. The automation is helpful, but for many code review processes you want to require reviews *before* check-in, not *after*. Like over-the-shoulder reviews, email pass-arounds are fairly easy to implement. They also work just as well across the hall or across an ocean.

A unique advantage of email-based review is the ease in which other people can be brought into conversations, whether for expert advice or complete deferral. And unlike over-the-shoulder, emails don't break developers out of "the zone" as they are working; reviews can be done whenever the reviewer has a chance.

The biggest drawback to email-based reviews is that they can quickly become an unreadable mass of comments, replies, and code snippets, especially when others are invited to talk and with several discussions in different parts of the code. It's also hard to manage multiple reviews at the same time. Imagine a developer in Hyderabad opening Outlook to discover 25 emails from different people discussing aspects of three different code changes he's made over the last few days. It will take a while just to dig though that before any real work can begin.

Another problem is that there's no indication that the review is "done." Emails can fly around for any length of time. The review is done when everyone stops talking.

So:

- Pro: Fairly easy to implement
- Pro: Works with remote developers
- Pro: SCM system can initiate reviews automatically
- Pro: Easy to involve other people
- Pro: Doesn't interrupt reviewers
- Con: Usually no verification that defects are really fixed
- Con: How do you know when the review is "complete?"
- Con: Impossible to know if reviewers are just deleting those emails
- Con: No metrics or process measurement/improvement

Pair-programming (review)

Most people associate pair-programming with XP and agile development in general. Among other things, it's a development process that incorporates continuous code review. Pair-programming is two developers writing code at a single workstation with only one developer typing at a time and continuous free-form discussion and review.

Studies of pair-programming have shown it to be very effective at both finding bugs and promoting knowledge transfer. And some developers really enjoy doing it. (Or did you forget that making your developers happy is important?)

There's a controversial issue about whether pair-programming reviews are better, worse, or complementary to more standard reviews. The reviewing developer is deeply involved in the code, giving great thought to the issues and consequences arising from different implementations. On the one hand, this gives the reviewer lots of inspection time and a deep insight into the problem at hand, so perhaps this means the review is more effective. On the other hand, this closeness is exactly what you don't want in a reviewer; just as no author can see all typos in his own writing, a reviewer too close to the code cannot step back and critique it from a fresh and unbiased position. Some people suggest using both techniques — pair-programming for the deep review and a follow-up standard review for fresh eyes. Although this takes a lot of developer time to implement, it would seem that this technique would find the greatest number of defects. We've never seen anyone do this in practice.

The single biggest complaint about pair-programming is that it takes too much time. Rather than having a reviewer spend 15-30 minutes reviewing a change that took one developer a few days to make, in pair-programming you have two developers on the task the entire time.

Of course pair-programming has other benefits, but a full discussion of this is beyond the scope of this article.

So:

- Pro: Shown to be effective at finding bugs and promoting knowledge-transfer
- Pro: Reviewer is "up close" to the code so can provide detailed review
- Pro: Some developers like it
- Con: Some developers don't like it
- Con: Reviewer is "too close" to the code to step back and see problems
- Con: Consumes a lot of up-front time
- Con: Doesn't work with remote developers
- Con: No metrics or process measurement/improvement

Tool-assisted review

This refers to any process where specialized tools are used in all aspects of the review: collecting files, transmitting and displaying files, commentary, and defects among all participants, collecting metrics, and giving product managers and administrators some control over the workflow.

"Tool-assisted" can refer to open-source projects, commercial software, or home-grown scripts. Either way, this means money — you're either paying for the tool or paying your own folks to create and maintain it. Plus you have to make sure the tool matches your desired workflow, and

not the other way around.

Therefore, the tool had better provide many advantages if it is to be worthwhile. Specifically, it needs to fix the major problems of the foregoing types of review with:

Automated File-Gathering: As we discussed in email pass-around, developers shouldn't be wasting their time collecting "files I've changed" and all the differences. Ideally, the tool should be able to collect changes *before* they are checked into version control *or after*.

Combined Display: Differences, Comments, Defects: One of the biggest time-sinks with any type of review is in reviewers and developers having to associate each sub-conversation with a particular file and line number. The tool must be able to display files and before/after file differences in such a manner that conversations are threaded and no one has to spend time cross-referencing comments, defects, and source code.

Automated Metrics Collection: On one hand, accurate metrics are the only way to understand your process and the only way to measure the changes that occur when you change the process. On the other hand, no developer wants to review code while holding a stopwatch and wielding line-counting tools. A tool that automates the collection of key metrics is the only way to keep developers happy (i.e., no extra work for them) and get meaningful metrics on your process. A full discussion of review metrics and what they mean (and don't mean) will appear in another article, but your tool should at least collect these three things: kLOC/hour (inspection rate), defects/hour (defect rate), and defects/kLOC (defect density).

Workflow Enforcement: Almost all other types of review suffer from the problem of product managers not knowing whether developers are reviewing all code changes or whether reviewers are verifying that defects are indeed fixed and didn't cause new defects. A tool should be able to enforce this workflow at least at a reporting level (for passive workflow enforcement) and at best at the version control level (with server-side triggers).

Clients and Integration: Some developers like command-line tools. Others need integration with IDE's and version control GUI clients. Administrators like zero-installation web clients and Web Services API's. It's important that a tool supports many ways to read and write data in the system.

If your tool satisfies this list of requirements, you'll have the benefits of email pass-around reviews (works with multiple, possibly-remote developers, minimizes interruptions) but without the problems of no workflow enforcement, no metrics, and wasting time with file/difference packaging, delivery, and inspection.

It's impossible to give a proper list of pros and cons for tool-assisted reviews because it depends on the tool's features. But if the tool satisfies all the requirements above, it should be able to combat all the "cons" above.

So what do I do?

All of the techniques above are useful and will result in better code than you would otherwise have.

To pick the right one for you, start with the top of the list and work your way down. The first few are the simplest, so if you're willing to live with the downsides, stop there. Tool-assisted review has the most potential to remove downside, but you'll have to commit to a trial period, competitive analysis, and possibly some budget allocation.

No matter what you pick, your developers will find that code review is a great way to find bugs, mentor new hires, and share information. Just make sure you implement a technique that doesn't aggravate them so much that they revolt.

More Java and Software Testing and Quality Resources

Article: Software Inspections

Software Testing Magazine

Software Quality Assurance Portal