Five Types of Review

Pros and cons of formal, over-the-shoulder, e-mail pass-around, pair-programming, and tool-assisted reviews.

There are many ways to skin a cat. I can think of four right off the bat. There are also many ways to perform a peer review, each with pros and cons.

Formal inspections

For historical reasons, "formal" reviews are usually called "inspections." This is a hold-over from Michael Fagan's seminal 1976 study at IBM regarding the efficacy of peer reviews. He tried many combinations of variables and came up with a procedure for reviewing up to 250 lines of prose or source code. After 800 iterations he came up with a formalized inspection strategy and whom to this day you can pay to tell you about it (company name: Fagan Associates). His methods were further studied and expanded upon by others, most notably Tom Gilb and Karl Wiegers.

In general, a "formal" review refers to a heavy-process review with three to six participants meeting together in one room with print-outs and/or a projector. Someone is the "moderator" or "controller" and acts as the organizer, keeps everyone on task, controls the pace of the review, and acts as arbiter of disputes. Everyone reads through the materials beforehand to properly prepare for the meeting.

Each participant will be assigned a specific "role." A "reviewer" might be tasked with critical analysis while an "observer" might be called in for domain-specific advice or to learn how to do reviews properly. In a Fagan Inspection, a "reader" looks at source code only for comprehension – not for critique – and presents this to the group. This separates what the author intended from what is actually presented; often the author himself is able to pick out defects given this third-party description.

When defects are discovered in a formal review, they are usually recorded in great detail. Besides the general location of the error in the code, they include details such as severity (e.g. major, minor), type (e.g. algorithm, documentation, data-usage, error-handling), and phase-injection (e.g. developer error, design oversight, requirements mistake). Typically this information is kept in a database so defect metrics can be analyzed from many angles and possibly compared to similar metrics from QA.

Formal inspections also typically record other metrics such as individual time spent during pre-meeting reading and during the meeting itself, lines-of-code inspection rates, and problems encountered with the process itself. These numbers and comments are examined periodically in process-improvement meetings; Fagan Inspections go one step further and requires a process-rating questionnaire after each meeting to help with the improvement step.

A Typical Formal Inspection Process

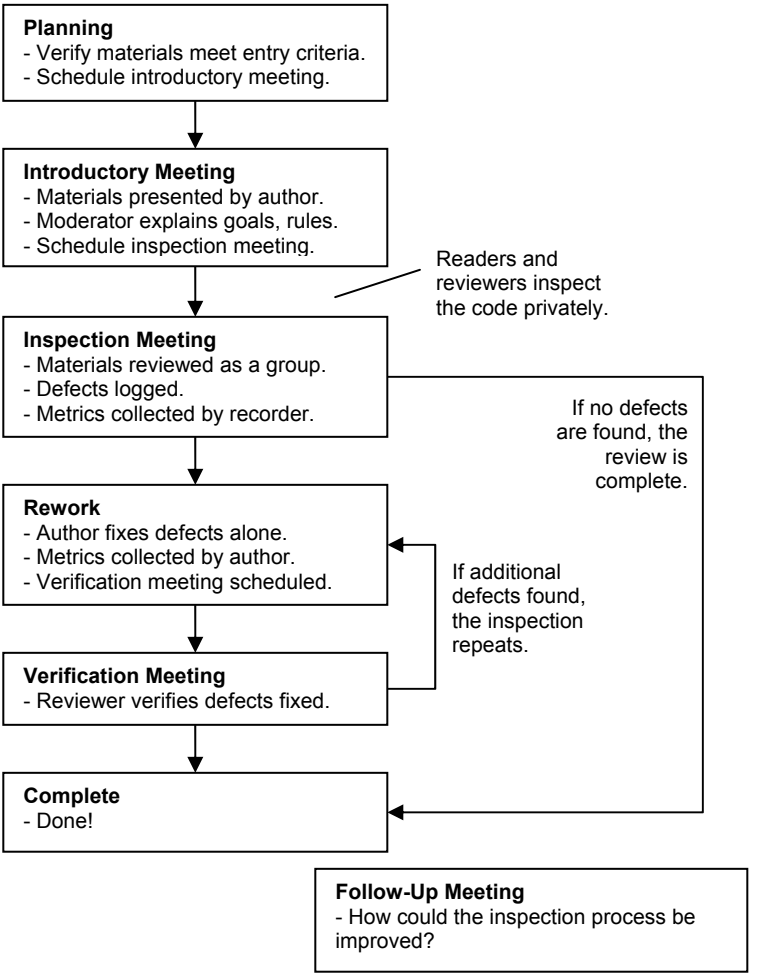

Figure 1: Typical workflow for a "formal" inspection. Not shown are the artifacts created by the review: The defect log, meeting notes, and metrics log. Some inspections also have a closing questionnaire used in the follow-up meeting.

Formal inspections' greatest asset is also its biggest drawback: When you have many people spending lots of time reading code and discussing its consequences, you are going to identify a lot of defects. And there are plenty of studies that show formal inspections can identify a large number of problems in source code.

But most organizations cannot afford to tie up that many people for that long. You also have to schedule the meetings – a daunting task in itself and one that ends up consuming extra developer time[1]. Finally, most formal methods require training to be effective, and this is an additional time and expense that is difficult to accept, especially when you aren't already used to doing code reviews.

Many studies in the past 15 years have come out demonstrating that other forms of review uncover just as many defects as do formal reviews but with much less time and training[2]. This result – anticipated by those who have tried many types of review – has put formal inspections out of favor in the industry.

After all, if you can get all the proven benefits of formal inspections but occupy 1/3 the developer time, that's clearly better.

So let's investigate some of these other techniques.

Over-the-shoulder reviews

This is the most common and informal of code reviews. 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.

Typically the author "drives" the review by sitting at the keyboard and mouse, opening various files, pointing out the changes and explaining why it was done this way. The author can present the changes using various tools and even run back and forth between changes and other files in the project. If the review sees

[1] See the Votta 1993 case study detailed elsewhere in this collection.
[2] See the case study survey elsewhere in this collection for details.

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. This complicates the process because you need schedule these sharing meetings and communicate over the phone. Standing over a shoulder allows people to point, write examples, or even go to a whiteboard for discussion; this is more difficult over the Internet.

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.

As with all in-person reviews, over-the-shoulders lend themselves to learning and sharing between developers and gets people to interact in person instead of hiding behind impersonal email and instant-messages. You naturally talk more when you can blurt out and idea rather than making some formal "defect" in a database somewhere.

Unfortunately, the informality and simplicity of the process also leads to a mountain of shortcomings. First, this is not an enforceable process – there's nothing that lets a manager know whether all code changes are being reviewed. In fact, there are no metrics, reports, or tools that measure anything at all about the process.

Second, it's easy for the author to unintentionally miss a change. Countless times we've observed a review that completes, the author checks in his changes, and when he sees the list of files just checks in he says "Oh, did I change that one?" Too late!

Over-the-Shoulder Review Process

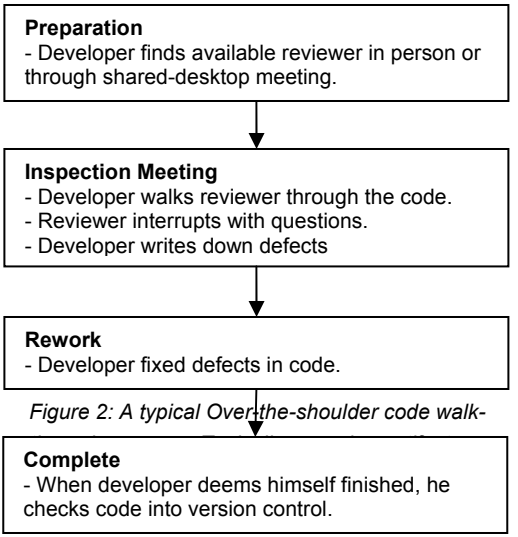

Figure 2: A typical Over-the-shoulder code walk-

Third, when a reviewer reports defects and leaves the room, rarely does the reviewer return to verify that the defects were fixed properly and that no new defects were introduced. If you're not verifying that defects are fixed, the value of finding them is diminished.

There is another effect of over-the-shoulder reviews which some people consider to be an advantage but others a drawback. Because the author is controlling the pace of the review, often the reviewer is lead too hastily through the code. The reviewer might not ponder over a complex portion of code. The reviewer doesn't get a chance to poke around in other source files to confirm that a change won't break something else. 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.

For example, say the author was tasked with fixing a bug where a portion of a dialog was being drawn incorrectly. After wrestling with the Windows GUI documentation, he finally discovers an undocumented "feature" in the draw-text API call that was causing the problems. He works around the bug with some new code and fixes the problem. When the reviewer gets to this work-around, it looks funny at first.

"Why did you do this," asks the reviewer, "the Windows GUI API will do this for you."

"Yeah, I thought so too," responds the author, "but it turns out it doesn't actually handle this case correctly. So I had to call it a different way in this case."

It's all too easy for the reviewer to accept the changes. But the next developer that reads this code will have the same question, and might even remove the work-around in an attempt to make the code cleaner. "After all," says the next developer, "the Windows API does this for us, so no need for this extra code!"

On the other hand, not all prompting is bad. With changes that touch many files it's often useful to review the files in a particular order. And sometimes a change will make sense to a future reader, but the reviewer might need an explanation for why things were changed from the way they were.

Finally, over-the-shoulder reviews by definition don't work when the author and reviewer aren't in the same building; they probably should also be in nearby offices. For any kind of remote review, you need to invoke some electronic communication. Even

with desktop-sharing and speakerphones, many of the benefits of face-to-face interactions are lost.

E-mail pass-around reviews

This is the second-most common form of informal 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 e-mail. Reviewers examine the files, ask questions and discuss with the author and other developers, and suggest changes.

The hardest part of the e-mail 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 e-mail and generate differences between each.

The version control system can be of some assistance. Typically that system can report on which files have been altered and can be made to extract previous versions. Although some people write their own scripts to collect all these files, most use commercial tools that do the same thing and can handle the myriad of corner-cases arising from files in various states and client/server configurations.

The version control system can also assist by sending the e-mails out automatically. For example, a version control server-side "check-in" trigger can send e-mails depending on who checked in the code (e.g. the lead developer of each group reviews code from members of that group) and which files were changed (e.g. some

files are "owned" by a user who is best-qualified to review the changes). The automation is helpful, but for many code review processes you want to require reviews before check-in, not after.

E-Mail Pass-Around Process: Post Check-In Review

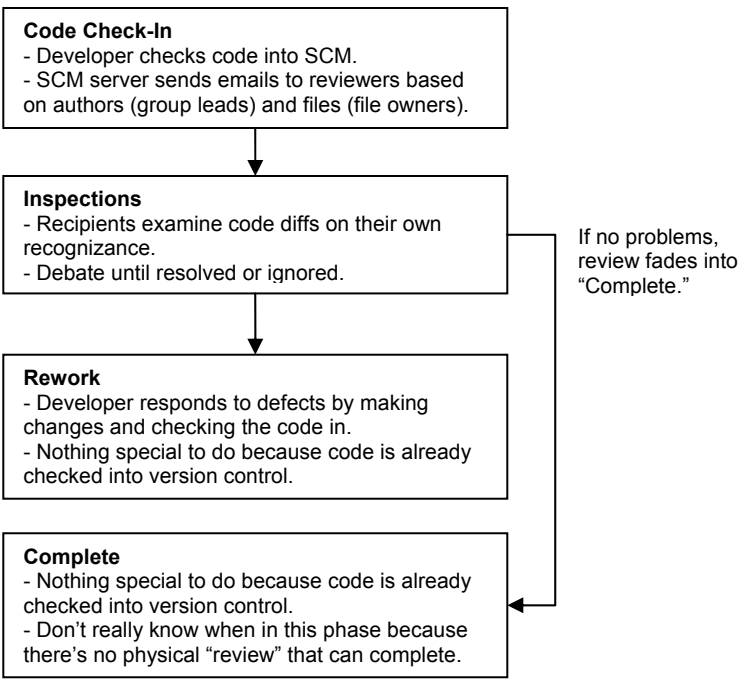

Figure 3: Typical process for an e-mail pass-around review for code already checked into a version control system. These phases are not this distinct in reality because there's no tangible "review" object.

Like over-the-shoulder reviews, e-mail pass-arounds are easy to implement, although more time-consuming because of the file-gathering. But unlike over-the-shoulder reviews, they work equally well with developers working across the hall or across an ocean. And you eliminate the problem of the authors coaching the reviewers through the changes.

Another unique advantage of e-mail pass-arounds is the ease in which other people can be brought into the review. Perhaps there is a domain expert for a section of code that a reviewer wants to get an opinion from. Perhaps the reviewer wants to defer to another reviewer. Or perhaps the e-mail is sent to many people at once, and those people decide for themselves who are best qualified to review which parts of the code. This inclusiveness is difficult with in-person reviews and with formal inspections where all participants need to be invited to the meeting in advance.

Yet another advantage of e-mail pass-arounds is they don't knock reviewers out of "the zone." It's well established that it takes a developer 15 minutes to get into "the zone" where they are immersed in their work and are highly productive[3]. Even just asking a developer for a review knocks him out of the zone – even if the response is "I'm too busy." With e-mails, reviewers can work during a self-prescribed break so they can stay in the zone for hours at a time.

There are several important drawbacks to the e-mail pass-around review method. The biggest is that for all but the most trivial reviews, it can rapidly become difficult to track the various threads of conversation and code changes. With several discussions concerning a few different areas of the code, possibly inviting other developers to the fray, it's hard to track what everyone's saying or whether the group is getting to a consensus.

---

[3] For a fun read on this topic, see "Where do These People Get Their (Unoriginal) Ideas?" Joel On Software. Joel Spolsky, Apr 29, 2000.

E-Mail Pass-Around Process: Pre Check-In Review

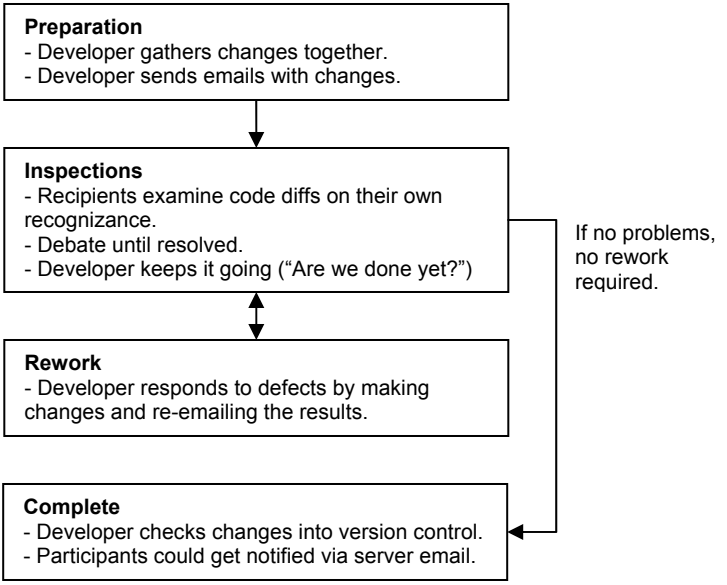

Figure 4: Typical process for an e-mail pass-around review for code already checked into a version control system. These phases are not this distinct in reality because there's no tangible "re-view" object.

This is even more prominent with over-seas reviews; ironic since one of the distinct advantages of e-mail pass-arounds is that they can be done with remote developers. An over-seas review might take many days as each "back and forth" can take a day, so it might take five days to complete a review instead of thirty minutes. This means many simultaneous reviews, and that means even more difficulties keeping straight the conversations and associated code changes.

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.

For all their advantages over over-the-shoulder reviews, e-mail pass-arounds share some disadvantages. Product managers are still not sure whether all code changes are being reviewed. Even with version control server-side triggers, all you know is that changes were sent out – not that anyone actually looked at them. And if there was a consensus that certain defects needed to be fixed, you cannot verify that those fixes were made properly. Also there are still no metrics to measure the process, determine efficiency, or measure the effect of a change in the process.

With e-mail pass-arounds we've seen that with the introduction of a few tools (i.e. e-mail, version control client-side scripts for file-collection and server-side scripts for workflow automation) we were able to gain several benefits over over-the-shoulder reviews without introducing significant drawbacks. Perhaps by the introduction of more sophisticated, specialized tools we can continue to add benefits while removing the remaining drawbacks.

Tool-Assisted reviews

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.

There are several key elements that must be present in a review tool if it is going to solve the major problems with other types of review[4]:

[4] In the interest of full-disclosure, Smart Bear Software, the company that employs the author of this essay, sells a popular peer code review tool called Code Collaborator for exactly this purpose. This product is described in a

Automated File Gathering

As we discussed in the e-mail pass-around section, you can't have developers spending time manually gathering files and differences for review. A tool must integrate with your version control system to extract current and previous versions so reviewers can easily see the changes under review.

Ideally the tool can do this both with local changes not yet checked into version control and with already-checked-in changes (e.g. by date, label, branch, or unique change-list number). Even if you're not doing both types of review today, you'll want the option in the future.

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 review code while holding a stopwatch and wielding line-counting tools.

A tool that automates 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 appears in another essay, but your tool should at least collect these three rates: kLOC/hour (inspection

---

different essay in this collection; this section will discuss general ways in which tools can assist the review process.

rate), defects/hour (defect rate), and defects/kLOC (defect density).

Review 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 that enforce workflow at the version control level).

Clients and Integrations

Some developers like command-line tools. Others prefer integrations with IDE's and version control GUI clients. Administrators like zero-installation web clients. It's important that a tool supports many ways to read and write data in the system.

Developer tools also have a habit of needing to be integrated with other tools. Version control clients are inside IDE's. Issue-trackers are correlated with version control changes. Similarly, your review tool needs to integrate with your other tools – everything from IDE's and version control clients to metrics and reports. A bonus is a tool that exposes a public API so you can make customizations and detailed integrations yourself.

If your tool satisfies this list of requirements, you'll have the benefits of e-mail 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.

The drawback of any tool-assisted review is cost – either in cash for a commercial tool or as time if developed in-house. You

also need to make sure the tool is flexible enough to handle your specific code review process; otherwise you might find the tool driving your process instead of vice-versa.

Although tool-assisted reviews can solve the problems that plague typical code reviews, there is still one other technique that, while not often used, has the potential to find even more defects than standard code review.

Pair-Programming

Most people associate pair-programming with XP[5] and agile development in general, but it's also 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.

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

[5] Extreme Programming is perhaps the most talked-about form of agile development. Learn more at http://www.extremeprogramming.org.

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.

Some developers just don't like pair-programming; it depends on the disposition of the developers and who is partnered with whom. Pair-programming also does not address the issue of remote developers.

A full discussion of the pros and cons of pair-programming in general is beyond our scope.

Conclusion

Each of the five types of review is useful in its own way. Formal inspections and pair-programming are proven techniques but require large amounts of developer time and don't work with remote developers. Over-the-shoulder reviews are easiest to implement but can't be instantiated as a controlled process. E-mail pass-around and tool-assisted reviews strike a balance between time invested and ease of implementation.

And any kind of code review is better than nothing.