Peering Into the White Box: A Testers Approach to Code Reviews

Alan Page
alanpa@microsoft.com

Abstract

Code reviews (including peer reviews, inspections and walkthroughs) are consistently recognized as an effective method of finding many types of software bugs early – yet many software teams struggle to get good value (or consistent results) from their code reviews. Furthermore, code reviews are mostly considered an activity tackled by developers, and not an activity that typically falls within the realm of the test team. Code reviews, however, are an activity that questions software code; and many testers who conduct code reviews question the software code differently than their peers in development.

This paper will present how a test team at Microsoft used code reviews as a method to improve code quality and more importantly as a learning process for the entire test team. The paper will also discuss how smart and consistent application of lightweight root cause analysis and the creation of code review checklists led the path to success – and how any team can use these principles to reach these same levels of success.

Biography

Alan Page began his career as a tester in 1993. He joined Microsoft in 1995, and is currently a Principal SDET on the Office Communicator team. In his career at Microsoft, Alan has worked on various versions of Windows, Internet Explorer, and Windows CE, and has functioned as Microsoft's Director of Test Excellence. Alan writes about testing on his blog (http://angryweasel.com/blog), is the lead author on How We Test Software at Microsoft (Microsoft Press, 2008), and contributed a chapter to Beautiful Testing (O'Reilly, 2009).

1. Introduction

Code reviews - the practice of systematically examining software source code - has been a long used practice in software development. Code reviews are one of the first lines of defense to find errors and are a “cheap” way to find bugs because of this. A second pair of eyes on anything is often one of the most straightforward ways to find an error. In my experience, code quality often improves if the author merely knows the code will be reviewed, as the author is less likely to write “tricky” code or take shortcuts.

Code reviews have a learning aspect as well. They work well for sharing techniques or algorithms, and are a great way to integrate new people onto the team or help team members learn about other parts of the project.

Finally, code reviews help a team build trust. Teams that conduct code reviews get used to the idea of analyzing the *code*, and not the person; and they get used to their *code* being critiqued and not themselves, and this gives them trust and freedom to talk about other things to improve in the product without making it a people issue.

2. The Tester Point of View

Code reviews are most often recognized as an activity done by the development team before checking in code or before passing it off in executable form to the test team. Developer code reviews are one of the farthest upstream activities there are to find bugs, thus are one of the most cost-effective methods of finding bugs.

Testing - even in the most agile of development environments, most often occur downstream from where developer code reviews happen. One notable exception is to pair a developer and a tester in a pair-programming session. Pair-programming is, in fact, a form of code review.

So why have testers perform code reviews?

On the Office Communicator team at Microsoft, our reasoning is this: our developers (and hopefully yours too) write unit tests. Unit tests are another upstream method of finding errors early and do a lot to improve code quality. However, we still test the code - on a much larger scale than unit tests are designed to do, but *our approach to testing from the test team perspective is different from the developer approach to writing unit tests.* Many unit tests, as good as they are at finding errors, test little more than the happy path. Unit tests test the code from the developer’s point of view and try to prove that the design and behavior is correct. Testers, on the other hand, ask numerous questions about the software they’re testing. Many of these questions begin with “what if?” – as in “what if we tried a really long string in this edit field?”, or “what if I try to launch the application twelve times?”. Most of these questions are well beyond the scope of what a unit test should do, and is one of the reasons we have software testers.

We apply the same reasoning to code reviews. When developers review code, they don’t ask as many questions. They make sure variable initialization is correct, that algorithms are optimal, and that the code structure and flow makes sense. The effort of review is worth it. On our team, many of the developer code reviews of significance (more than a few dozen lines) finds at least one issue or suggestion during review.

We also hypothesize that testers will ask different questions of the code than developers. When reviewing code, instead of asking questions like “will this code work correctly?” testers are more prone to ask questions like “under what situations that I can think of will this code fail?” The tester code reviews were not designed under any means to replace the developer reviews already in place, but were designed as

just another test technique for the test team. With this in mind, we set out on our experiment of tester code reviews.

3. Types of Code Reviews

Code reviews vary in style, formality, and effectiveness. The least formal type of review is where the author of the code literally grabs someone out of the hall and says, “Hey – would you take a look at this?” A step up from there (in both formality and effectiveness) is to send an email or instant message to teammates asking for a quick review. At the other end of the spectrum are formal code inspections where pre-work for each reviewer is identified, and a moderator ensures that the code is reviewed, and statistics are tracked carefully in a group meeting.

Microsoft development teams use varying degrees of formality in code reviews – typically falling somewhere between the extremes outlined above. Code reviews are a good practice for development teams and their value to product quality is significant.

Code inspections as invented by Michael Fagan[i] are the most formal method of code review, and are highly effective at finding errors in code early in the product development cycle. However, they require that developers spend a large amount of their time preparing for, and attending the inspection meetings (also in training to know how to do a Fagan inspection in the first place). In many organizations practicing Fagan inspections, the number of total person-hours required for reviews will outweigh the coding time. This is, in my experience, the primary reason more teams do not attempt Fagan inspections.

4. Our Approach to Tester Code Reviews

For our tester code reviews, we decided to try a slightly less formal approach. We wanted the reviews to be as thorough as possible while keeping time investment as low as possible and interest and engagement of the team high.

4.1 The Kickoff

We began the effort by presenting a tech talk for the test team on code reviews. Since most of the team was only familiar with ad-hoc code reviews, we presented an overview of different code review approaches and techniques in order to provide some background information.

A week later, we held another kickoff meeting for all testers on the team who were interested in taking part in this code review experiment. We did not look for any specific characteristics or skills in the participants other than a willingness to try something new and commit a small amount of time every week to the effort (note that nearly all testers at Microsoft are fluent programmers). In this meeting, we went over the details of our code review process and expectations for the volunteers. The expectations defined at the beginning of the experiment included:

- **Review for 60-90 minutes each day, and record the time spent on review.** Part of the learning process for doing good code reviews is practice. By establishing a rhythm of performing reviews, we expect testers to improve their technique. However, we’ve also found that the ability to find errors consistently in code declines after 60-90 minutes, and we also wanted each participant to discover when their own reviewing abilities started to fade within a session. Finally, we asked them to track the actual time so we could calculate yield – the number of issues they found within their review period.

- **Make a note of all issues or questions found during the review.** We used this information in the review meeting, as well as for tracking yield.
- **Schedule a regular meeting for team reviews.** We asked teams to schedule a regular meeting where they would discuss and note errors or questions found during the individual reviews. We had two primary reasons for the regular meetings. The first was to determine which issues multiple members of the team are finding, as these had a higher likelihood for being real errors that are likely to be fixed. The second, and equally as valuable was the learning aspect. The differences in issues found by different team members lead to discussion and sharing of how a particular reviewer discovers an error, which in turn leads to improvement by all reviewers in subsequent reviews.
- **Use a checklist.** We gave everyone a checklist including types of errors generally found during code review. We asked them to select one item on the checklist, and review a section of code *looking only for that single item*. (A "section" could be a single function, a class, a screen, or any other sensible partition). Looking for a single item enables a reviewer to minimize distractions and find errors more consistently. Reviewers would then have the option of either reviewing an additional section of code looking for the same class of error, or could review the same section again looking for a different error. A sample checklist is included in the appendix.

At the end of the kickoff meeting, we divided the attendees into teams (mostly representing particular feature areas) and gave each team the initial assignment of selecting the code for their first review.

4.2 Selecting the Code for Review

We did not have a goal of reviewing all code. Given the size of our product and the other testing investments we have, this was not feasible or practical. Instead, we used other factors to guide us in this decision.

We used a risk-based approach to identify potential code for review. Some of the areas we looked at to identify risk included:

- **Code Churn** – We identified areas of the product with high amounts of recently modified or added code and selected source files within that area as potentially reviewable.
- **Code Complexity** – Code with a large number of paths or variables, or code with a high number of dependencies is prone to error.
- **Tester Intuition** – "Gut feel" or "tester intuition"; testers often have a hunch of where bugs may lurk, and in our experience, these hunches are accurate enough to influence the selection.

5. Tester Code Reviews in Action

During each meeting, we began by noting how much time each reviewer was able to review the code. Next, we walked sequentially through the code assignment for the week, stopping to discuss each issue noted by the reviewers. These discussions included bugs as well as areas of confusion (e.g. details of the .Net common language runtime garbage collector). The result of much of the initial effort was in learning the product, or some of the idiosyncrasies of the programming languages (our product uses both C++ and C#).

Most teams averaged about one thousand lines of reviewed code each week. The initial stages included a lot of learning time as the reviewers got used to the idea of reviewing the code and using the checklist. We also encouraged everyone to suggest new checklist items if they found additional errors, and to

suggest removing checklist items that were not (or did not seem) effective at finding errors. We were not trying to find every single possible error, and keeping the number of checklist items manageable was important.

Testers found, on average, ten unique “issues” per thousand lines of code during review. Slightly less than ten per cent of these were bugs that could affect the customer experience of the product. The remainder of the issues were code issues potentially affecting maintenance and readability of the code. The intangible value of a maintainable code base is important enough, that over eighty-five per cent of these issues are scheduled to be addressed.

6. Lessons Learned

Test team code reviews remain in progress on our team, and we continue to tweak our approach as we learn what works and what does not work. Thus far, test team members as well as product management view the approach as successful, and as something that should continue throughout this release and into the next release of our product.

Some of the biggest lessons learned include:

- **Learning** – Initially, much of the code review effort went toward learning the product. The testers were adept in the functionality of the product, but reviewing the code opened up many of the nuances of the implementation. Although we didn’t find many bugs in the initial steps, we felt that learning the how and why of the code implementation was a benefit for the team. Several of the testers on the team found additional areas or parameters to test based on the code review.
- **Balancing the code review commitment with “other” test activities** – we also discovered that it was difficult for many of the reviewers to commit regularly to the 60-90 minutes of review per day. Each of the testers involved in the review was also responsible for their day-to-day testing work, and this “new” effort sometimes fell by the wayside.
- **Involving the development team** – we started the process attempting to have as little development effort as possible, but we changed that approach slightly. Originally, the test team performed the reviews, and then one representative from test took the results to the developer and met one on one to discuss the issues and questions. We ended up modifying that and invited the developers to attend the review meetings. This enabled us to transfer knowledge between test and development faster, and increased the ability of testers to read the code.
- **Developing specialization and experts** – reviewing code is a lot different from most other test techniques, so we did not expect everyone to become an expert. Over time, testers tend to specialize, and we expected that there would be differentiation in tester’s ability to review code effectively. In the relatively short time, we’ve asked testers to review code we’ve seen three of the participants stand out as effective code reviewers. We expect that, as we continue with this effort, that a few more stars will rise.
- **The checklist approach** – the formality of the approach worked well for the team. The use of checklists and the inspection meeting encouraged learning and participation, without adding too many extra meetings. Our goal was to keep the approach as simple and lightweight as possible within a small level of formality (e.g. checklist driven inspections, regular meetings, and a few metrics).
- **Bug tracking** – we remain excited about the learning opportunities that these reviews created, but another potential drawback of our current approach is that we do not track the bugs found during code review in our product bug tracking system. We did this purposely to keep the system lightweight, but by doing that we lose a bit of insight that we can share with future testers and

developers on the team about how the code works and what types and numbers of bugs have been found in a particular component over time from reviews.

7. Recommendations

Code reviews performed by testers have been successful on our team, and we expect that the practice will work well for any team with some obvious precursors. The need for source code access is obvious, but a level of trust between the testers and developers on the team is also necessary in order to be successful. Testers need to know that their feedback will be taken seriously, and developers need to understand that the critique is of the code, and not a critique of the developer. The testers will also need enough programming language background to be effective in reviewing the code.

8. Future Plans

We expect to continue the practice of test reviews for the long term. It has enabled our test team to find bugs in a new way and has facilitated learning across the team. In addition, we'll ask our "star" reviewers to examine much of the code introduced late in the product cycle to reduce the risk of regression or new bugs from those late changes. We'll also continue to encourage and coach teams in reviewing code so that we can continue to grow new review experts and increase the quality of our product code overall.

We will also examine how we can tie the comments and feedback from the reviews to the actual code. We expect that this will make learning the code much easier for new team members, and since we're only looking at code that we deem to be risky in the first place, chances are that any additional context we can provide with the code will be beneficial for anyone involved in developing or testing that code.

9. Recommended Reading

Software Inspection – Tom Gilb & Dorothy Graham. Addison-Wesley Professional (January 10, 1994)

The Best Kept Secrets of Peer Code Review - http://smartbear.com/codecollab-code-review-book.php

Software Inspection (web article) - http://www.the-software-experts.de/e_dta-sw-test-inspection.htm

Fagan Inspections (Wikipedia entry) - http://en.wikipedia.org/wiki/Fagan_inspection

Appendix

Sample checklist items

- Are input parameters validated and checked for null?
- All exceptions must not be caught and sunk (i.e. not rethrown)
- Do the comments accurately describe the code?
- Examine loops for performance issues
- There must be no hard-coded strings and magic numbers in the code
- All user-visible strings must be localizable (i.e. retrieved from a resource file)
- Verify correctness of operators
- Is arithmetic safe for overflow and 64-bit math?
- Are the functions or source files exceptionally large or complex?
- Are COM interfaces properly ref counted?
- If there are multiple exit points (returns) from a function, can they be combined?
- Is the code overly complex? Could it be simpler?
- Do the function and variable names make sense?
- All memory allocations should be checked for failure
- Are shared resources guarded appropriately using synch objects?
- Are threads cleaned up on exit?
- Check for precedence errors – parenthesis are free
- Are untrusted inputs validated?
- Check for properly constructed switch statements
- Are comments appropriate and useful?
- Does the code have performance implications – if so, are optimizations possible?
- Are there spelling errors in names or comments?
- Have boundary conditions been checked?
- Are there unused elements in data structures?
- Are string buffers big enough for localized content?
- Does the new code duplicate code that exists elsewhere in the project?
- Are there unnecessary global variables?

[i] http://en.wikipedia.org/wiki/Fagan_inspection