BASIL VANDEGRIEND: PROFESSIONAL SOFTWARE DEVELOPMENT

Strategies for Effective Code Reviews

Posted by Basil Vandegriend on September 16th, 2007. Categories: Coding Tags: code review, Coding, quality, software development

Code reviews are an important practice for improving the quality of your software and ensuring that it is ready for release. Software engineering research has found that reviews (or inspections) are a powerful QA practice and have many advantages over testing:

- A higher percentage of defects are found when using reviews – as high as twice as many than testing.
- When an issue is detected by a review, the root cause of the problem is usually known. Defects found by testing require further work to analyze.
- Reviews can be performed earlier in the development life-cycle than testing.
- Reviews provide more precise and more immediate feedback for developers to learn and improve from than testing.

Steve McConnell in his book Rapid Development summarizes these findings on pages 73 to 75.

Given these many benefits, it is disappointing that more teams do not use code reviews. If you are part of a team that does regular code reviews, then I congratulate you. Just because you do code reviews, however, does not guarantee that you will see all the benefits. In fact, I have often observed highly ineffective reviews. One example from my own experience is when a developer was asked to do a review but viewed the task as an annoying interruption from more interesting work. They did a very quick and superficial 'review', and then reported that everything looked fine. As a task the review was completed, but the expected quality improvements were not achieved. In a sense the review did not really happen.

How can you avoid this from happening to you and your team? In this article I present a set of strategies for achieving effective code reviews. These are based on my years of experience performing code reviews and figuring out what works the best. Some of these ideas are inspired by concepts from PhotoReading, a multi-step system for efficiently extracting desired information from written text (aka reading, but not as it is traditionally done).

Adopt a Critical yet Playful Attitude

The primary purpose of a code review is to critique the code - to find defects, potential problems, or simply poorly-written sections – which requires a critical mindset that is different from the typical problem-solving mindset used by developers as they solve code. You cannot simply read over a section of code and accept what you see: you need to challenge the assumptions and decisions behind it. This is especially important for finding errors of omission. It is much harder to find a defect because of what was not written, and to do so requires you have a questioning, suspicious attitude towards the code you are reviewing. You need to mentally prepare yourself before starting a review by reminding yourself of the purpose of the review and the need for a critical, questioning attitude.

Maintaining this critical attitude throughout your review, however, can be difficult, especially if it is not your regular mode of thinking. It helps to cultivate a delight or enjoyment in finding issues with the code, like you are playing a game searching for treasures hidden by an opponent. Unearthing a serious defect should feel equivalent to hitting a home run in baseball. Viewing the review as a fun yet challenging game where you score points for the issues you find will boost your motivation and keep you focused in that critical mindset.

The critical attitude required for reviews is similar to the questioning mindset needed for root cause analysis. In root cause analysis, however, you are trying to determine why something failed, whereas in a code review you are trying to find all potential causes of failure.

Take Multiple Passes Through the Code

In order to do an effective review you must learn how the code is structured, what it is trying to accomplish, and where it falls short. It is difficult to do this all at once as you read through the code for the first time. A much more effective approach is to use multiple passes to review the code with the initial passes focused on learning about how the code works and later passes focused on critiquing the code.

The number of passes you will need will depend on the size of the code base you are reviewing, its complexity, and how thorough a review you are doing. As a starting point, here is a suggested approach for reviewing a code base in multiple passes.

1. *Problem Overview*: Review the requirements or specification to understand what the code is supposed to do. What problem is it trying to address? What requirements is it implementing?
2. *Solution Overview*: Review the design document or specification to learn the overall solution to the problem. How are the requirements being addressed?
3. *High Level Organization*: Review the code or detailed design document to figure out how the code is organized and functions at a high level. What are the packages and the major classes within each? I find UML class diagrams showing class relationships and hierarchies very useful for this pass, and if none already exist as part of the documentation I often end up sketching out such a diagram. UML sequence or state diagrams can also be useful to highlight the main interactions or behavior of the system.
4. *Major Class Review*: Review the code for the most important classes as identified previously. While issues and problems can be found in earlier passes through the code, particularly at the design level, this pass marks the transition from learning about the code to critically analyzing it for problems.
5. *Multiple Detailed Review Passes*: The previous passes will have given you a sufficient understanding of the code base to critically review the remainder of the code base. Even at this point, however, it is still valuable to use multiple passes. I explain why in the next section.

Focus on One Goal Per Pass

One common response to the problem of poor-quality reviews is the creation of a review checklist that lists all the issues a reviewer should be looking

for. Often these checklists enumerate many specific items and go on for more than a page. Two examples of checklist items are: (1) checking the return codes of system calls, and (2) using prepared SQL instead of SQL literals for parameters. While the idea of checklists has merit, the reviewer is given no guidance as to how to effectively use the checklist. Short-term memory, which can only hold approximately seven pieces of information, is all needed for analyzing the code. Trying to remember a page of issues to look for while reviewing simply does not work.

The solution to this is to focus on a single goal for each detailed review pass through the code. Each goal can be as specific as an individual checklist item, but I find this level of detail is seldom necessary and leads to too many passes. The goals I prefer to use generally correspond to categories of issues. For example, instead of a checklist item such as checking return codes of system calls, I have the goal of reviewing error handling. As I make my review pass with this goal in mind, I can ignore the bulk of the logic and focus instead on how errors might occur and how they are handled.

Unlike the long checklist, this solution is workable because you only need to hold a single piece of information – the current review goal – in your short-term memory while you are reviewing. Focusing on a single goal, however, provides an additional benefit: it actually improves our ability to find problems in the code pertaining to the goal. How does this work? Our eyes and other senses constantly transmit a wealth of information to our brain – too much information, in fact, to fully process in detail. The brain must therefore selectively filter the information it receives. The brain decides what information to select based in part on our goals and desires – what we view as important. In essence, our thoughts and perspective affects how we view the world. When we focus on a single goal such as error handling, as we review the code the brain is more likely to focus on segments of code dealing with error handling (or the lack thereof). This allows you to review the code faster: you can skim along, skipping past sections that do not apply, then slow down to carefully analyze a relevant block or a suspicious section, and then speed back up again. Changing your goal between review passes causes your perspective to shift, allowing you to identify a different set of issues in the same code base. This greatly increases the effectiveness of the review by catching more issues across a wider variety of categories.

Here are some typical goals I use for my reviews:

- *Functionality*: Does the code meet the business (functional) requirements? Is the logic correct?
- *Class Design*: Does each class have low coupling and high cohesion? Is the class too large or too complicated?
- *Code Style*: Is duplication of code avoided? Are any methods too long? Are typical coding idioms / standards followed?
- *Naming*: Are packages, classes, methods and fields given meaningful and consistent names?
- *Error Handling*: How are errors dealt with? Does the code check for any error conditions that can occur?
- *Security*: Does the code require any special permissions to execute outside the norm? Does the code contain any security holes?
- *Unit Tests*: Are there automated unit tests providing adequate coverage of the code base? Are these unit tests well-written?

Add Review Comments Directly to the Code

How should issues found during the review be reported back to the original developer? One commonly used approach is to create a review document listing all the issues found. For each issue the problematic section of code must be referenced, usually by supplying the file name and line number. Management often prefers the review document because it can be used for process auditing and generating metrics. Exclusive use of a review document for recording issues, however, causes problems for the reviewer and the original developer. For the reviewer, switching from the code to a separate document can easily break their flow, especially since they must record not just the issue, but also the reference back to the code. For the original developer, once they start fixing issues in a particular file, the references (file name and line number) of other issues in the same file can quickly become out of date, especially if refactorings such as move method, extract method, or rename class are performed.

I believe, therefore, that the most effective approach to recording issues is to add them directly to the code base as comments, at the place the issue occurs. Each such comment should be tagged with a special label – I use `REVIEW` – so that the issues can be easily found later by the original developer. In fact, if you use the Eclipse IDE for Java development, you can define a task tag for this type of comment that will cause all such comments to appear in the Task view. Once the review is completed, the code with the review comments can be checked back into the version control system to be made available to the original developers. If you must produce a review document to comply with your organization's processes, then I would recommend creating the document at the end of the review from the review comments you added to the code.

Summary

Code reviews are a powerful practice for improving software quality. The strategies I have found personally useful in increasing the effectiveness of my reviews are:

- Adopt a critical yet playful attitude.
- Take multiple passes through the code.
- Focus on one goal per pass.
- Add review comments directly to the code.

If you have any additional strategies you have found particularly useful, please let me know.

3 Comments on "Strategies for Effective Code Reviews"

1. [...] Professional Software Development » Strategies for Effective Code Reviews "You need to mentally prepare yourself before starting a review by reminding yourself of the purpose of the review and the need for a critical, questioning attitude." Nice article on how to make code reviews effective. (tags: code review software development programming practice) [...]

   Written by Infovore » links for 2007-10-08 on October 8th, 2007 at 4:20 pm.

2. Thank You !!!!!!!!!

   Written by Arun on February 5th, 2009 at 4:22 am.

3. This is really nice article. It should also explain with examples so that it will become more effective

   Written by Pravin Sharma on February 12th, 2009 at 3:58 am.
