Social Effects of Peer Review

Unexpected positive social aspects; handling hurt feelings, and the "Big Brother Effect."

Perhaps a manager's most difficult task is to deal with emotions and human interactions. It's easy to think of Vulcan-like developers having no feelings, but nothing could be farther from the truth, outward appearances notwithstanding.

Any criticism is an opportunity both for growth and for embarrassment. In our experience with customers and with in-house code review we've uncovered several social issues that managers and team-leads should be aware of. Some are positive and should be encouraged in the group; others are negative and need to be addressed in a way that is both sensitive and effective.

The "Ego Effect"

The first effect takes place even before the first code review happens. The ego will inflict both good and bad feelings in code review; in this case good.

"Jerry always forgets to check for NULL-pointer exceptions." No one wants to be known as the guy who always makes silly, junior-level mistakes. So imagine yourself in front of a compiler, tasked with fixing a small bug, but knowing that as soon as you say "I'm finished" your peers – or worse your boss – will be critically examining your work.

How will this change your development style? As you work – certainly before you declare code-complete – you'll be a little more conscientious. Do your comments explain what you did and why? Did you remember to check for invalid input data? Are there are few more unit tests you could write?

In other words, you become a better developer *immediately*.

You're going to do a great job because you want a perfect review report. "Wow," your boss will declare in the meeting notes, "I've never reviewed code that was this thoughtful! I can't think of a single thing to say! Perfect!"

OK, maybe not. But you want the general timbre of behind-your-back conversations to be "Oh yeah, his stuff is pretty tight. He's a good developer" and not "He's pretty good but makes a lot of silly errors. When he says he's done, he's not."

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 you have a 1-in-3 chance of being called out for review, that's still enough of an incentive to make sure you do a great job. There is a breaking point, however. For example, if you just had a 1-in-10 chance of getting reviewed, you might be sloppier because now you have an

excuse. "Yeah, I usually remember to do that. You just caught me on a bad day."

This is an example of the ego working for us. Later we'll see how the ego can result in social problems, but for now let's stay positive and investigate a fun, productive aspect of peer code review that is rarely pointed out.

Old Habits Die Easy

It was our second code review at Smart Bear Software using an alpha version of our new code review tool. I was reviewing a small change Brandon made to some Java code. He had added the following line of code:

```
if ( "integrate".equals( str ) ) { ... }
```

I stared at it for a moment. He was testing whether a string `str` was equal to the constant string `integrate`. But I would have written it the other way – with `integrate` and `str` switched. I realized both methods work, but perhaps there was a reason he picked this one?

I shot him a quick message. He explained that if `str` is null, his way the expression will return `false` whereas with my way we'd have a null-pointer exception. So he's in the habit of putting constant strings on the left side of an `equals()` expression like that.

Good trick! A new habit I can get into that will eliminate an entire class of bugs from my code[1]. I just became a slightly better developer. And I had fun doing it!

---

[1] As a C developer I was used to the similar trick of putting numeric constants on the left side of double-equal signs in conditions, e.g. if(0==x) rather than if(x==0). This way if I accidentally used a single-equal operator the compiler would complain. The Java example is analogous, and yet somehow my brain didn't make the connection. It

Later we were talking about the review and we realized two totally unexpected things happened here.

First, my learning experience had nothing to do with the bug Brandon was fixing. Just the fact that we were communicating about source code meant we were also sharing knowledge.

Second, I was the reviewer, yet I was the one learning something. Normally you think of the review process as unidirectional – the reviewer points out problems for the author, and the author might learn something in the process. But here I was the reviewer, and yet I was the one who was learning!

This was not an isolated incident. In the next four weeks we had all learned a lot from each other – everything from programming tricks to obscure facts about the system API and virtual machine. Our collective insights were spread around quickly – far faster that any other technique we could have invented.

And the best part was: It was fun! It's just plain fun to learn something new and to grow as a developer. And it's fun when everyone else in the office is learning too – it's just as fun to teach as to learn and you get the feeling that the whole development organization is accelerating.

Systematic Personal Growth

It gets even better.

Try this experiment sometime[2]: For one week, make a log of every error you make. Every misspelling in e-mail, every time you accidentally close the application you're working in, every time you have a bug in your code, even every syntax error you make. Don't

---

just goes to show you're never too old to learn something in computer science!

[2] These ideas were inspired by the Software Engineering Institute's (SEI) Personal Software Process (PSP). This is a collection of methods for increasing personal productivity in concrete, measurable ways.

get too specific; keep each item in the list high-level and use hash marks to count how many times you make that type of mistake.

For now, just imagine you've done this. As you might guess, certain patterns emerge. A few of the items have an awful lot of hash marks next to them. In fact, by the end of the second day you might have been getting tired of counting some of them!

And that annoyance might live in the back of your mind. You might even start thinking about it consciously. Pretty soon you'll anticipate the mistake and prevent yourself from making it. Some of those will come easy, others will take work to eliminate.

But eventually you develop habits that prevent that type of error completely. If you frequently forget to close parenthesis in your code, perhaps you'll enable the feature in your code editor that closes parenthesis for you. If you close applications by accident, maybe you'll pause just a few seconds every time you're about to hit the close button. Maybe you'll look up how to spell common words or learn how to input your common mistakes into the auto-correct facility in your e-mail editor.

Over time you become more productive, more efficient. You're working faster and smarter. Not through some self-help seven-step program, just by observing yourself a little more carefully and systematically than you're used to.

Code review is an opportunity to do exactly this with regard to your software development skills. Reviewers will do the work of determining your typical flaws; all you have to do is keep a short list of the common flaws your work. Over time, correct them. Pick a few at a time to work on. No pressure, no deadlines, just think about it sometimes.

In fact, this technique works almost as well if you review your own code than when someone else reviews it. Usually you need to at least view the code in a format that isn't your standard editor to get the problems to jump out at you, just as errors in Word

documents become clear only when printed instead of viewing on-screen. Indeed, many code inspection advocates insist that inspections must be done only with print-outs for this very reason.

Even if employed informally and irregularly, this technique helps you become a better, more efficient developer by leveraging the feedback you're already getting from peer review. And once again, becoming better is fun!

But of course not all social aspects of code review are fun and personally rewarding. Managers should be aware of these problems and know how to mitigate them.

Hurt Feelings & The "Big Brother" Effect

There are two major ways in which code review adversely affects the social environment of the development team. Fortunately it turns out that one managerial technique can address both problems at once. Here we explain the two problems and give specific ways for managers to address them.

First: Hurt feelings.

No one takes criticism well. Some people blow up more easily than others, but everyone feels a twinge when defects are pointed out. Especially when the subject is trying his hardest, deficiencies pointed out can feel like personal attacks.

Some people can handle it, correct the problem, laugh at their own foibles, and move on, where others takes things personally and retreat to their cubicle to ruminate and tend to their bruised ego.

Second: The "Big Brother" effect.

As a developer you automatically assume it's true. Your review metrics are measured automatically by 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 is the basis of process improvement, but metrics can be used for good or evil. If developers believe their metrics will be used against them, not only will they be hostile to the process, but you should also expect them to behave in a way that (they believe) improves their metrics, rather than in a way that is actually productive.

Of course these feelings and attitudes cannot be prevented completely, but managers can do a lot to mitigate the problem. It can be difficult to notice these problems, so managers have to be proactive.

Singling someone out is more likely to cause more problems than it solves. Even in a private one-on-one conversation it's difficult to receive any kind of advice that might temper emotions.

We recommend that managers deal with this by addressing all developers as a group. During any existing meeting, take a few moments to assuage everyone at once using some of the points below. Don't call a special meeting for this – then everyone is uneasy because it seems like there's a problem. Just fold it into a standard weekly status meeting or some other normal procedure.

These points are all ways of explaining that (a) defects are good, not evil and (b) defect density is not correlated with developer ability and that therefore (c) defects shouldn't be shunned and will never be used for performance evaluations.

1. Hard code has more defects

Having many defects doesn't necessarily mean the developer was sloppy. It might be that the code itself was more difficult – intrinsically complex or located in a high-risk module that demands the highest quality code. In fact, the more complex the code gets

the more we'd *expect* to find defects, no matter who was writing the code.

Indeed, the best way to look at this is to turn it around: If you knew a piece of code was complicated, and a reviewer said he found no flaws at all, wouldn't you suspect the reviewer of not being diligent? If I presented this essay to an editor, even after multiple passes at self-review, and the editor had no comments at all, shouldn't I suspect the editor of not having read the text?

It's easy to see a sea of red marks and get disheartened; your good hard work is full of flaws. But remember that difficult code is supposed to have flaws; it just means the reviewers are doing their jobs.

2. More time yields more defects

An interesting result of studies in code inspection is that the more time the reviewer spends on the code the more defects are found. In retrospect that might sound obvious – the longer you stare at something the more you can say about it. But the amount of time needed to find all defects is greater than you might guess.

In some cases, the group tried to target a certain amount of time per page of code, testing whether spending 5, 15, 30, or even 60 minutes per page made a difference. Most found increases in defect densities up until at least 30 minutes per page, some even saw increases at 60 minutes per page. After a certain point the number of defects would level off – at some point you really have considered everything you can think of – and further time spent on the code would yield no extra defects. But 60 or even 30 minutes per page is a lot more time than most people would guess.

But what this also means is that the quantity of defects found in a piece of code has more to do with how much time the reviewer spent looking at the code as opposed to how "good" the code was to begin with. Just as more complex code ought to yield

more defects, reviewers that spend more time on code ought to uncover more defects. And this still has nothing to do with the author's productivity or ability.

3. It's all about the code

Remind everyone that the goal of this process is to make the code as good as possible. All humans are fallible, and mistakes will be made no matter how careful or experienced you are.

In 1986 the space shuttle Challenger exploded 110.3 seconds after lift-off, killing all seven crew members. The subsequent presidential investigation sought to answer two questions: What caused the explosion, and why wasn't this prevented?

The first question was answered in a famous demonstration by the Nobel Prize-winning physicist Richard Feynman[3]. The answer wasn't incredible; the rubber O-rings used to connect sections of the fuel task were inelastic at low temperatures and therefore allowed fuel to leak out the side of the tank where it subsequently ignited. The troubling part is that this wasn't news. This was a documented, well-understood physical property. So why didn't they cancel the launch?

Suddenly the second question became much more interesting. Not just to figure out whom to blame for the debacle, but to correct systemic problems that impact the safety of all missions.

It got worse. It turns out the rocket engineers *did* report the data that implied that launching at 29 degrees Fahrenheit was unsafe. The data were contained in a report given from rocket engineers to launch officials the night before the launch.

So the launch officials were at fault for authorizing a launch when engineers had data showing it was unsafe? No, because the

[3] Feynman dunked a rubber O-ring in a glass of ice water and demonstrated that they became brittle at that temperature – the same temperature as it was at the launch.

manner in which the data were presented completely hid the trend that predicts this effect[4]. The raw information was present, but the charts were drawn and data arranged in a manner that completely obscured the key relationship between O-ring failure and launch temperature. Given that presentation, no one would have concluded the launch was unsafe.

So the engineers were at fault for misrepresenting the data? No, because the data were correct. No one thought to re-arrange the data in a manner that would have shown the trend. No one was trained in data visualization and presentation techniques.

The take-home point is this: Even with many, very intelligent and experienced people, mistakes will be made. NASA addressed this problem in nine parts, most of which are forms of review[5]. The point of software code review is the same – to eliminate as many defects as possible, regardless of who "caused" the error, regardless of who found it. This isn't personal.

Indeed, as a manager you know that if "quantity of defects" *were* used in performance evaluations, developers would have incentive to not open defects even when they know they're there. That's the opposite of the goal of the review process, so you know that defects must necessarily not factor into any personal – or personnel – report.

Tell your developers you understand what they're afraid of; then they're more likely to believe that their fears are unfounded.

---

[4] A fascinating description of the problem (and a solution that would have prevented this disaster) is given in Edward Tufte's *Visual Explanations: Images and Quantities, Evidence and Narrative.*

[5] See the official correspondence between NASA and the Presidential Commission on the Space Shuttle Challenger Accident. http://history.nasa.gov/rogersrep/genindex.htm

4. The more defects the better

The preceding points lead to a seemingly illogical conclusion: The more defects we find, the better. After all, if finding a defect means a Challenger disaster is averted, than finding defects is good. If we'd expect defects in important or complex code, finding defects in that code is good. If reviewers naturally find more defects when they're diligent, higher numbers of defects are good.

This conclusion is in fact correct! Think of it this way: Every defect found and fixed in peer review is another bug a customer never saw. Another problem QA didn't spend time tracking down. Another piece of code that doesn't incrementally add to the unmantainablility of the software. Both developers and managers can sleep a little easier at night every time a defect is uncovered.

The reviewer and the author are a team. Together they intend to produce code excellent in all respects, behaving as documented and easy for the next developer to understand. The back-and-forth of code-and-find-defects is not one developer chastising another – it's a process by which two developers can develop software of far greater quality than either could do alone.

The more defects that are found, the better the team is working together. It doesn't mean the author has a problem; it means they're both successfully exploring all possibilities and getting rid of many mistakes. Authors and reviewers should be *proud* of any stretch of code where many defects were found and corrected.

Also, as described earlier, defects are the path to personal growth. As long as over time you're eliminating your most common types of mistakes, defects are a necessary part of the feedback loop that makes you a better developer. So again, the more defects that are found the better; without that feedback you cannot continue to grow.