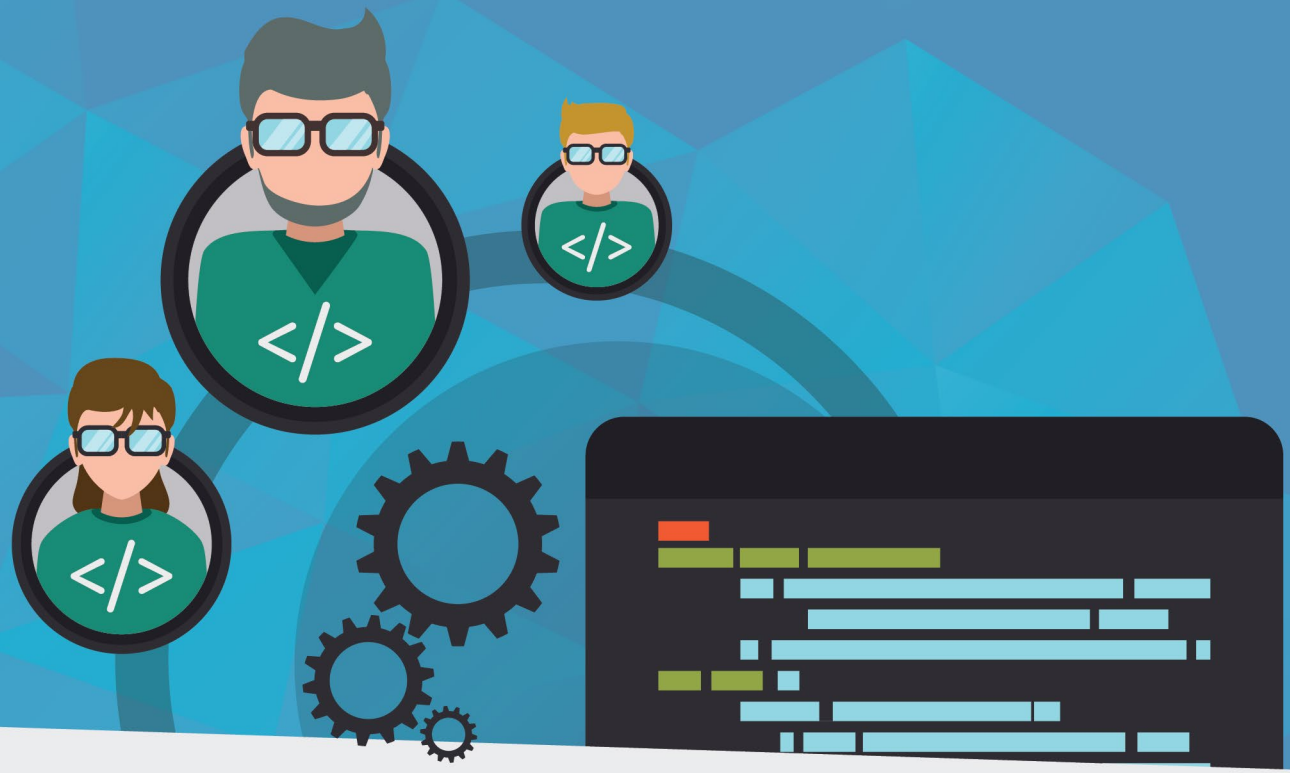
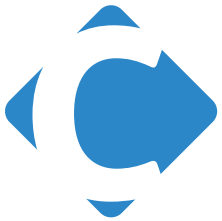


THE COMPLETE GUIDE TO

APPROACHING YOUR DEVELOPMENT TEAM WHEN THEY WRITE BAD CODE

Collaborator



Collaborator

Collaborator is a code review tool that **helps development, testing and management teams work** together to produce high quality code.

LEARN MORE ABOUT **COLLABORATOR**

Content

You're sitting at your desk, trying to "track" down a bug that's been reported, when *it* happens. The hunt takes you into some method that inspires you to do a double take. It's about 1,200 lines long, it has switch statements nested three deep, and you think (but you aren't sure) that it does the same thing two or three times in a row for no particular reason. You look at the source control history and see that this is another "Bob special." After seeing this, you start thinking about finally having a long overdue talk with Bob so that you don't have to keep cleaning up these messes. That sure won't be a fun talk. *So how do you approach it?*

We've All Seen Bad Code

Let's be clear about something up front. We've all encountered a developer's bad code at some point in our careers. If it wasn't their's it was our own code. Getting really good at telling teammates that their code is littered with problems is like getting really good at breaking into your car after locking yourself out of it: it's tactically useful in the moment but indicative that you need a better overall strategy. Your goal shouldn't be to master gently telling coworkers about their bad code but rather to make the mastery unnecessary. And I say that not as some kind of meta cop-out, but rather to put your strategy into context as an attempt to start or further a relationship.

> *"Your goal shouldn't be to master gently telling coworkers about their bad code but rather to make the mastery unnecessary"*

When you're part of a team, someone on your team who is committing bad code is a failure of everyone on the team—yourself included. So as you prepare for the intervention you're planning with the person

in question, keep in mind that you aren't some kind of neutral crime scene investigator, sizing things up antiseptically. You're part of the problem, and you share in the responsibility. Your team. Your code. Your problem.

The good news is that if you approach this conversation constructively, you're taking the first step toward fixing the problem, the code, and thus the team. The key is making it constructive.

> *"You are part of the problem, and you share in the responsibility. Your Team. Your Code. Your Problem"*

5 Ways to Make Code Criticism Constructive

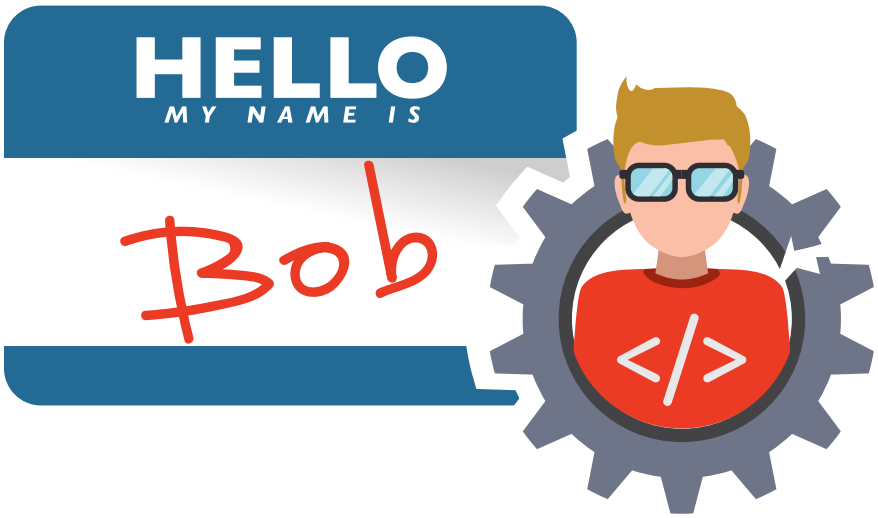
1. **Don't have the conversation when you're frustrated or angry.** Instead, wait until you're calm and rational.

2. **Don't get into this unless there's a demonstrable problem.** If you and he just have different casing preferences, the tension you create is probably going to nullify the benefits of standardization. Cosmetic coding standards and other relatively minor concerns can and should be addressed with automated static analysis.

3. **Don't rely on seniority or status in any way.** There's no faster way to breed resentment than forcing people to do things they don't agree with "because you say so."

4. **Don't expect to revolutionize someone's entire approach in a single sitting and don't make the conversation a marathon affair.** You want to have a clear and relatively concise message so that you get your point across without exhausting the other person. Improvement will happen over the time.

5. **Finally, don't say that the code is "bad."** That's a useless, subjective way to categorize. Everything in software is about trade offs. What you want to do is show Bob that he's paying for quick and dirty coding with maintenance headaches for the rest of the team.



Build A Constructive Code Strategy and Environment

You've already prepared a bit by reading what not to do, so now it's time to complement that with what to do. There needs to be three main components to this preparation:

(1) the gaps you want to address

(2) the support for your argument

(3) the outcome for which you're hoping

These three components are going to frame the discussion you intend to have. The gaps are actual, specific problems with Bob's code.

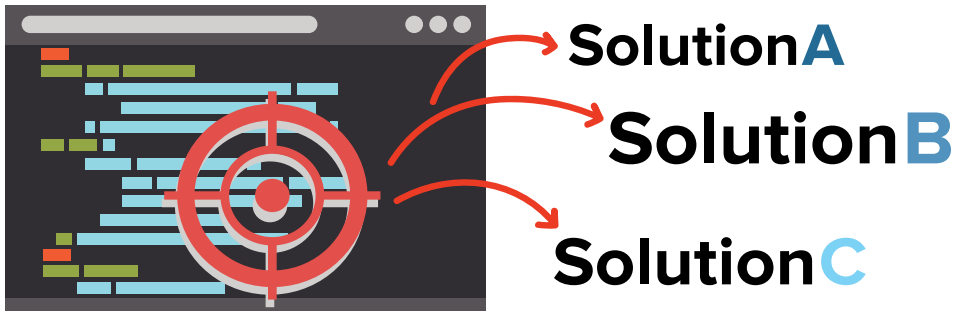
You don't want to stroll over to Bob's desk, pull up a chair, sit on it backwards and say, "So, Bob, you're pretty bad at this programming thing...great talk!" You need to decide what tangible items you want to address during this discussion. What's the most egregious source of problems? Is it the gigantic methods? The nested switch statements? The duplicate code? Pick one or maybe two of these things to cover. Just as you don't want to be critical and vague, you also don't want to be critical and devastatingly specific, reading off 95 of Bob's greatest coding flaws like some kind of departmental Martin Luther. But if you want to fix all of them, it's important to lay the groundwork for a mentoring relationship because you're definitely not going to fix all of them in one day.

> *"Just as you don't want to be critical and vague, you also don't want to be critical and devastatingly specific"*

Building Support: Do Your Research.

Let's say that you've decided to focus on method length as the topic to address. The next thing to do is **build support for your argument.** It's a lot more credible to cite some supporting studies or widely respected industry figures on the matter than to march over to Bob and declare that his methods are too long. Build a case with evidence for the principle that you want to cover and then also find specific, problematic instances in the code base to discuss. The last thing you want is to be hand-wavy about the problem—you want to be able to point at it and say, "for instance, this right here is a really big method."

> *"Build a case with evidence for the principle that you want to cover"*



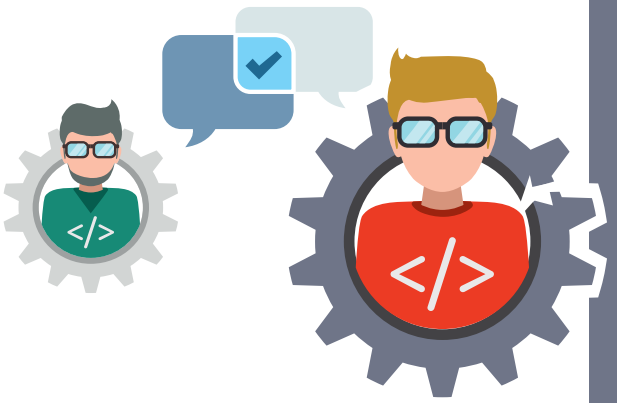
The Outcome Should Be Actionable

Having picked your issue and built a case, the last thing to do is choose an outcome toward which to steer the conversation. So you've shown Bob a giant method that he wrote and convinced him of the evils of giant methods. He'll say, **"So what now?"** Decide ahead of time that you want to work together to break the method into X number of smaller methods or that you want to leave the code in a state where no refactored method is longer than Y lines. Whatever it is, pick an actionable item so that you and Bob can cap the conversation off with a joint win.

Planning Your Approach

It's worth noting that you should never shame someone for their code. 'Code shamers' aren't helpful to your situation or your working environment. Developers will learn from their mistakes overtime if you guide them. Having a game plan in place prior to your conversation will help you handle the bad code situation professionally without completely crushing his/her dreams.

At this point, you're ready to have the hard conversation. If you do it right, it won't be nearly as hard as you might think, and it will serve as a productive starting point for a series of subsequent conversations that will be easier and perhaps even pleasant.



It's *Not* If You're Going to Say Something, But *How*

Having built your case ahead of time, it's time to go have a chat with Bob. Approach the situation with a calm and rational attitude. You've got legitimate argument and you are all set for a constructive dialog, but the lead in for the conversation threatens to be awkward if you don't approach it right.

What I'd suggest doing to put the conversation in more natural term is to ask for his help. Approach the conversation by saying you've found a bug during one of your code reviews. Instead of walking over to preach to Bob about the evils of his code rather ask him to help solve a problem with you because this method that you've found is a bit tough to follow.

One of the most effective ways to find surface code problems is through the Socratic Method. Instead of telling Bob that the method is too long, ask a series of questions;

"Wow, good thing you're here—this is a pretty long method with a lot going on. How long do you think it would take the average team member to understand it?"

"Huh, wow, three or four hours seems like a pretty long time to spend trying to understand a method, don't you think?"

Making proclamations of fact or strongly stating opinions tends to put people in a defensive posture. Again 'code shaming' is not the avenue to go down when having this conversation. Instead, ask questions. If you approach it from the side of curiosity, they tend to join you in problem-solving mode rather than argue against you in debate mode. (Still, asking questions this way may not lead to the desired outcome, which is why you've done your homework.) If you start to see this happening, switch gears from the questions to statements of your experience and how you feel. These are inarguable.

"I get that you think breaking up the method would just distribute the complexity, but I spent hours trying to understand this. I don't have the same problems in Alice's code where she uses lots of small methods."

There's nothing Bob can say, so far, to argue with this, so it's a good premise for a call to action.

"You can find a lot of literature and studies out there about small methods correlating with fewer bugs, and I know I'm not the only one on the team that prefers to read and write code that's more compact. What do you think? Can we try pairing up and refactoring this a bit? It might help reduce the defect count originating from features that you've implemented."

That last line is the critical last piece to the engagement puzzle. Bob probably can't argue with your well-researched position, and he certainly can't argue with how you feel, but he could still balk at changing his approach. You need to focus on how the new approach benefits not only others or the code base in general but also specifically Bob himself. **No one wants to write bad code and suffer the consequences, so help them understand that life can be better.**

Maintaining the Constructive Relationship

As I've said, you're not going to take someone that writes bad code and turn the whole ship around in a sitting. The first engagement is about establishing a relationship along these lines and setting the stage for more talks. To do that, you have to be prepared,

patient, willing to nudge gently but firmly, and opportunistic in looking for a line of reasoning that strikes a chord. If you can show Bob the difference and convince him of the benefits of adopting a new approach, future collaborative improvement sessions will start to happen naturally. Often times, they'll happen with Bob being eager to learn. I've personally been in a lot of groups where building trust in this fashion has led to unsolicited requests to review others' code and offer feedback. If that happens, you're in an excellent place.

Over the course of time you and Bob can work together on a broader strategy of code improvement. **Even if he's receptive and eager in the initial conversation, this kind of ongoing maintenance will be required for going from "He/she writes poor code" to "He/she writes clean code."** This will require cataloging the differences between good code and bad code, keeping track of them, and working strategically with Bob on one or two of them at a time to help him get better. As he practices the easier ones, you can gradually introduce more and more to gently push him out of his comfort zone, toward cleaner code.

So, in the end, how do you tell Bob that his code sucks?

In the best case scenario, you and Bob would have wthis conversation over a beer, a year or two later, when marveling at how far he's come:

"I didn't want to discourage you back then, but your code really sucked. You've come a long way. Good job."

But in this case, the best time to approach 'Bob', the developer who writes bad code is when you have a full-proof plan in place. The conversation won't go exactly as you initially planned but if you have the correct material and right attitude going in to the conversation, you'll be able to have a mature conversation about the issue at hand.

Measuring Developers

The structure of most organizations that employ software developers is this: the developers who are responsible for code report up through a structure of people that don't. To put it another way, the people responsible for personnel management usually don't understand how to evaluate the work of those reporting to them – at least not directly.

Even if people with titles such as "Director of Software Engineering" or "Development Manager" had been developers at some point, asking them to perform code reviews isn't a solid approach. It wouldn't scale well. Imagine asking someone to do detailed code reviews with seven or eight people while also doing all of the other things a manager is required to do, such as budgeting, dealing with other departments, worrying about software licensing, etc. And that's even assuming they're technical. Most managers have to squint pretty hard into their rear-view mirrors to see the last time they wrote a lot of code, if there ever was a time. This is where senior developers would be more useful.

Typically, managers will rely at least in part on results and observable behaviors. Does the developer keep long hours? Does the developer take one for the team? Did the developer heroically work 90 hours last week to get the code out on time? Did the developer fix a lot of bugs, or better yet, write code in which not a lot of bugs were reported?

All of that sounds reasonable until you think of technical debt. "Technical debt" is a term that describes a trade: when you take shortcuts in the code in order to ship today, it's at the cost of having a mess when you try to ship down the line. For a great visual of this, imagine a child tasked with cleaning his room who simply stuffs all of the clutter and food debris under the bed. In this light, the very developers that managers view as effective may be writing awful code and making a mess. Tired developers, working 90 hour weeks, are almost certainly thrashing in a tech debt cycle — making heroic efforts to overcome problems that they created in the first place by making a mess.

It's hard for a manager to make any form of direct evaluation, even when it seems as though there are easy ones to be made.

Don't Be Driven By Metrics

Okay, so they can't make direct evaluations. What about indirect ones? Well, those are tricky as well. One of the most common traps for software management, particularly if it's not terribly mature, is the allure of metrics. Perhaps you've heard calls for a team build that reports unit test coverage, method size, or cyclomatic complexity? If a manager could get access to those statistics, the reasoning goes he/she could know who on the team was writing good code and who wasn't.

Be careful what you measure. It's possible to achieve a high degree of unit test coverage by writing test methods that don't assert anything. You can keep method size under control by having classes with thousands of tiny methods. Perhaps the most iconic example of unintended consequences for measuring developer productivity is the mountains of terrible code that resulted from managers, at one time, trying to measure developer productivity in lines of committed code. Metrics are tempting for managers, but there be dragons. Relying on metrics to measure developer effectiveness has not historically been a path to success.

"Metrics are tempting for managers, but there be dragons. Relying on metrics to measure developer effectiveness has not historically been a path to success."

Measurement via Human

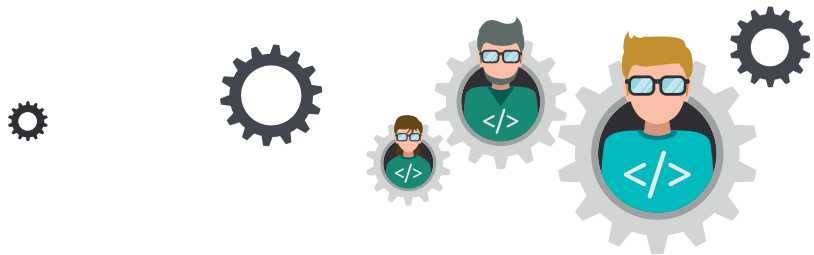
If the managers who make personnel decisions can't rely on themselves to evaluate developers and they can't rely on machines to do it, what choice is left? Clearly, they're going to have to turn to other humans to do this. Common patterns that you see are the appointment of a trusted advisor in the form of someone with a title like "Architect" or "Tech Lead." Or perhaps they take a more egalitarian approach, like having the developers perform peer reviews or pair program. In this fashion, the manager delegates evaluation to people in a position to do it, and she uses the information they furnish to make decisions.

But, going back to the original question, what if Bob–the one checking in the bad code–is a senior team member or even the architect? Think it's not possible? It happens all the time. A manager trusting an advisor will introduce one check and balance, but it's hardly foolproof. It gets better when there is regular peer review and pairing. More checks, more balances, more eyes, and more voices. Not only are bugs and defects being found but your more junior members are learning from senior developers through this process.

If we eliminate all evaluation options for managers with the exception of presiding over a team with extensive peer review, we're back to square one.

It's going to be up to a member of the team without any authority over Bob to let Bob know that he needs to improve his code. It's going to take conversations, persuasion, and collaboration, as opposed to management cracking the whip and sizing people up. In the end, there's only one reliable way for management to ensure that there aren't Bobs out there, writing bad code indefinitely without feedback. They need to create a collaborative culture of positive feedback and trust so that the team of humans they're managing take care of one another and spur each other on toward improvement.

Bob's bad code is the team's bad code, not management's. And the team, not management, needs to own the code.



With the **right tools and best practices**, your team can peer review all of your code.

SMARTBEAR

Over 3 million software professionals and 25,000 organizations across 194 countries use SmartBear tool

3M+ users

25K+ organizations

194 countries

See Some Succesful Customers >>

API READINESS

Functional testing through performance monitoring

SEE API READINESS PRODUCTS

TESTING

Functional testing, performance testing and test management

SEE TESTING PRODUCTS

PERFORMANCE MONITORING

Synthetic monitoring for API, web, mobile, SaaS, and Infrastructure

SEE MONITORING PRODUCTS

CODE COLLABORATION

Peer code and documentation review

SEE COLLABORATION PRODUCTS

Collaborator

SMARTBEAR