Chilling the Messenger

KEEPING EGO OUT OF SOFTWARE-DESIGN REVIEW

Software Design
**who is
KV?**

click for video

Dear KV,

I was recently hired as a midlevel web developer working on version 2 of a highly successful but outdated web application. It will be implemented with ASP.Net WebAPI. Our architect designed a layered architecture, roughly like Web Service > Data Service > Data Access. He noted that data service should be agnostic to Entity Framework ORM (object-relational mapping), and it should use unit-of-work and repository patterns. I guess my problem sort of started there.

Our lead developer has created a solution to implement the architecture, but the implementation does not apply the unit-of-work and repository patterns correctly. Worse, the code is really hard to understand and it does not actually fit the architecture. So I see a lot of red flags coming up with this implementation. It took me almost an entire weekend to work through the code, and there are still gaps in my understanding.

This week our first sprint starts, and I feel a responsibility to speak up and try to address this issue. I know that I will face a lot of resistance, just based on the fact that the lead developer wrote that code and understands it more than the alternatives. He may not see the issue that I will try to convey. I need to convince him and the rest of the team that the code needs to be refactored or reworked. I feel apprehensive, because I am

like the new kid on the block trying to change the game. I also don't want to be perceived as Mr. Know-It-All, even though I might be a little more opinionated than I should be sometimes.

My question is, how can I convince the team that there is a real problem with the implementation without offending anyone?

~Opinionated

Dear ~Opinionated,

Let me work backwards through your letter from the end. You are asking me, Kode Vicious, how to point out problems without offending anyone? Have you read any of my previous columns? Let's just start out with the KV ground rules: it's only the law and other deleterious side effects that keep me on the "right" side of violence in some meetings. I'd like to think a jury of my peers would acquit me should I eventually cross to the wrong side, but I don't want to stake my freedom on that. I will try my best to give you solutions that do not land you in jail, but I will not guarantee them not to offend.

Trying to correct someone who has just done a lot of work, even if, ultimately, that work is not the right work, is a daunting task. The person in question no doubt believes that he has worked very hard to produce something of value to the rest of the team, and walking in and spitting on it, literally or metaphorically, probably crosses your "offense" line—at least I think it does. I'm a bit surprised that since this is the first sprint and there is already so

In order to become more comfortable with the system, there are two things to call for: a design review and a code review.

much code written, shouldn't the software have shown up after the sprints established what was needed, who the stakeholders were, etc.? Or was this a piece of previously existing code that was being brought in to solve a new problem? It probably doesn't matter, because the crux of your letter is the fact that you and your team do not sufficiently understand the software in question to be comfortable with fielding it.

In order to become more comfortable with the system, there are two things to call for: a design review and a code review. These are not actually the same things, and KV has already covered how to conduct a code review ["Kode Reviews 101." *Communications of the ACM* 52(10): 28-29. (October 2009)]. Let's talk now about a design review.

A software design review is intended to answer a basic set of questions:

1. How does the design take inputs and turn them into outputs?
2. What are the major components that make up the system?
3. How do the components work together to achieve the goals set out by the design?

That all sounds simple, but the devil is in the level of the details. Many software developers and systems architects would prefer that everyone but themselves see the systems they have built as black boxes, where data goes in and other data comes out, no questions asked. You clearly do not have the necessary level of trust with the software you're working with to allow the lead developer to get away with that, so you should call for a design review

where you take the lid off the box and poke around at the parts inside. In fact, questions 2 and 3 are going to be your main tools for figuring out what the software does and whether or not it is suitable for the task.

When I have to interview people for jobs, I always ask them questions about systems they have worked on while we draw out the block diagram on a whiteboard: What are the major components? How does component A talk to component B? What happens if C fails? I'm trying to transfer their mental images of their software into my own mind, of course without either going mad or having a nasty flashback. Some pieces of software are best left outside your mind, but hopefully that's not going to be the case with the system you're working with.

Remember that every box that this person draws can be opened if you think you're not getting sufficient detail. Much like the ancient game show, "Let's Make a Deal," it is always OK for you to ask, "What's behind door number 1, Monty?" Of course, you might find that it's a goat, but hopefully you find that it's a working set of components that are understandable to you and the team.

The one thing *not* to do in a design review is turn it into a code review. You are definitely not interested in the internals of any of the algorithms, at least not yet. The only code you might want to look at are the APIs that glue the components together, but even these are best left abstract, so that the amount of detail does not overwhelm you. Remember that the goal is always to get the big picture rather than the fine details, at least in a design review.

The Socratic method can be applied in an annoyingly pedantic way, but since you're trying not to give offense, I suggest that you play by a few useful rules.

Coming back to the question of offense, I have found only one legal way to avoid giving offense, and that is always to phrase things as questions. Often called the Socratic method, this can be a good way to get people to explain to you, and often to themselves, what they think they are doing. The Socratic method can be applied in an annoyingly pedantic way, but since you're trying not to give offense, I suggest that you play by a few useful rules. First, do not hammer the person with a relentless list of questions right off. Remember that you are trying to explore the design space in a collaborative way; this is not an interrogation. Second, leave spaces for the people you're working with to think. A pause doesn't mean they don't know; in fact, it might be that they're trying to adjust their mental model of the system in a way that will be beneficial to everyone when the review is done. Lastly, try to vary the questions you ask and the words you use. No one wants to be subjected to a lot of, "And then what happens?"

Finally, I find that when I'm in a design review and about to do something that might give offense, such as throwing a chair or a whiteboard marker, I try to do something less obvious. My personal style is to take off my glasses, put them on the table and speak in a very calm voice. That usually doesn't offend, but it does get people's attention, which leads them to concentrate harder on working to understand the problem we're all trying to solve.

KV

Kode Vicious, known to mere mortals as George V. Neville-Neil, works on networking and operating system code for

fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, and rewriting your bad code [OK, maybe not that last one]. He earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts, and is a member of ACM, the Usenix Association, and IEEE. Neville-Neil is the co-author with Marshall Kirk McKusick and Robert N. M. Watson of The Design and Implementation of the FreeBSD Operating System (second edition). He is an avid bicyclist and traveler who currently lives in New York City.

Copyright © 2016 held by owner/author. Publication rights licensed to ACM.

SHAPE THE FUTURE OF COMPUTING!

Join ACM today at acm.org/join

**BE CREATIVE.
STAY CONNECTED.
KEEP INVENTING.**

Association for
Computing Machinery

Advancing Computing as a Science & Profession