Bringing Best Practices to a Long-Lived Production Code

"Best Practices for HPC Software Developers" webinar series
Date: January 17, 2018
Presented by: Dr. Charles R. Ferenbaugh of Los Alamos National Laboratory

---

Questions answered during webinar:

Q: Isn't the answer "getting along fine" an answer based entirely upon use and not at all upon development? If the same question was asked of the developers, the answer would certainly NOT be "getting along fine". So, isn't it the case that the question of the code's health cannot be answered solely by users.

A: If the users are able to do what they need to do now, they may think that the code is getting along fine. But if the developers are having a harder and harder time providing that same level of service, perhaps if they're losing people and the people coming on board can't give the same level of support, or if the existing team members are having a harder and harder time giving the level of support that the users need, then the users may not know it, but as the project continues, they won't be getting along fine any more either. So there's perhaps a lag as to when the users realize it, but in time the users will not be getting along fine either.

Q: Any suggestions on addressing the social side of the best practice start up inertia. Useful code was written with best practices of the time…

A: I'll say a bit more about that in the next section. *(See slides 9-11.)*

Q: When changing practices and codes, will the process by which best practices emerge be documented as well?

A: We've been using a project wiki, in our case it's Atlassian Confluence, to start documenting what are the practices we're doing, how are we changing them. And having that, as a central repository for the team to refer to, has been a big help. That's not something I thought to mention in the slides, but that has been a big part of what we're doing.

Q: How many developers on the xRage project and what funding source?

Q: What amount of funding was required to make the culture shift on the xRage project?

A: I don't know exactly, but it was a combination of new funding specifically for advanced architecture support, and redirection of some existing resources for the physics folks to support that. I'd guess about 3 FTEs on the CS side, and maybe 2 or 3 FTEs worth of the physicists' time being redirected. This is a project that had about 12 or 15 FTEs all together. That would be a good order-of-magnitude guess.

Q: It appears that the CS co-lead may have served the role of a CTO for example. What do you think?

Q: Would you argue that domain-science leadership should defer to CS leadership on issues related to software development? Is that a practical approach in your organization? If not, why?

A: We have a fairly good track record of the physics lead and the CS lead working together. The CS lead has primary input into the CS issues, and the physics lead has primary input into the physics issues, but they work together to balance the competing constraints on what to do with the code and the resources we have. From what I've seen that's been fairly successful.

Q: How did you generate this dependency graph?

A: I'll address that in a slide that coming up. *(See slide 19.)*

Q: Were there any particular tools (other than CS eyeballs) used to identify where these various kinds of changes would help?

A: I have a slide coming up that will address that. *(See slide 19.)*

Q: How do you confirm that code you remove is not being used anymore?

A: For the most part, our physicists are in enough contact with the user community that the physicist who knows a particular functionality has a pretty good sense of "this is being used, this is not." We sometimes supplement that with database tracking that, to a limited degree, tracks what features of the code are being used in production, to confirm that something is being used or not being used.

Q: Did feature development stop during all this cleanup time?

A: Feature development didn't stop, but it did slow some. Our management picked out a few key features that they thought were most important, and development on those went on in parallel with the work we're talking about here. But the number of features was smaller than it would have traditionally been.

Q: What are you using for your home-grown build system? GNU make?

A: It is GNU make, with a lot of home-grown infrastructure built around it.

Q: Were there issues with private branches of users that were used eg to develop new features?

A: One thing that's been key is that we have, over time, discouraged users from doing their own private builds of the code, and standardized on using either released versions, or in some cases HEAD versions of the master branch. Except in very rare cases, we've not given branches to the users. And in those cases, they were very friendly users who understood that we were doing something exceptional, and didn't do it for a long period of time.

Q: Was a rewrite ever considered instead of fixing/cleanup?

A: Yes, a rewrite was considered. There's actually been a lot of discussion on that point. When we were discussing that, we realized that writing a new code was a possibility, but it was a much harder task than many people realized.

Because in order to rewrite a code, to take over the functionality of a code that you have, you have to first figure out what the old code does, which in many cases is not at all obvious in a code that is convoluted and not well-documented. Then you have to write the code. Then you have to do all the V&V to prove that the new code does what the old one does, in all the cases that your users are interested in. And you have to do your V&V not only to your own satisfaction, but to the users' satisfaction as well. This was going to be a process of several years if we followed through with it, even in the best case. So the decision was made, taking all those things into account, to do a rewrite.

That was the decision we made for our project. I'm not going to say that's the best choice for all projects, but it is a decision where you really have to weigh everything, including the hidden costs of trying to write a new code.

Q: What is the justification for moving from GNU make to cmake? I agree with everything on this list, but our project's personal experience is that GNU make is reliable across virtually all architectures we've run on, whereas cmake/automake/etc. has platform (and often version) specific bugs that substantially impact users attempting to compile the code

A: GNU make is indeed reliable across all architectures. The infrastructure that we built on top of it was working fairly reliably, but it was so complex that there was only one person on the project who truly understood it. And there's the old danger of, if this person wins the lottery and decides to quit, the whole project is going to have a very hard time understanding the system to continue to do work. So the idea was, by making the transition to CMake, which has more standardized ways of writing of the functionality that some of this infrastructure has, that you have something that would be understandable by more people, and reduce that risk of being one person thin in running the build system.

Q: What if management supports the idea of change, but is unable to throw any money behind it?

A: As I said earlier, we had a combination of new money and redirection of existing money. I think if there's no new money available, you could still do the sort of thing we're talking about just by redirecting existing money, although the scope might have to be smaller and the timeline longer. I think it could still be done. If management thinks it's sufficiently important to redirect some resources, that can be done.

Q: Any tips when the push to SE best practices is a grassroots movement and not from the top down?

A: That's hard to make a general statement. The best indicator, I would say, is if aspects like planning, and staffing, and training and career development, if there's not movement to change those things, then the effort is likely to just stay at a grassroots level.

Q: Looking at the slides ahead I'm not sure if the social interaction between developers will be addressed further. Restating my first question: Are there any suggestions for commencing code clean up and integration of best practices into a long-lived code? How to approach experienced domain experts about refactoring for modern architectures and maintainability without alienating them?

A: What we've done on xRage is, we've recognized and been very clear that the domain experts are a very important part of the project. They know how the science in the code works, and they know what is needed to support the users, in a way that the computer scientists do not. What we want to do is leverage that expertise.

For instance, when doing code cleanup, that has often been done by a computer science person and a domain expert working together, or perhaps by the domain scientist having gotten some ideas from the computer science folks as to what sorts of things would be useful. We had some computer scientists take the lead in refactoring some of the infrastructure packages, and that served as an example that the domain scientists could look at.

The other thing I'd add is that, for advanced architectures, we have also tried to make that a joint effort between the CS and the domain science side. We perhaps have the CS people take a little bit more of the lead on that, but in some cases some of our domain scientists have worked with them, picked up what needed to happen, and taken on some of that work as well.

So It very much needs to be a collaborative process, and I think for the most part that has worked very well on both sides. Both sides have recognized the expertise of the other, and not felt threatened, because it's complementary expertise and not replacement expertise.

Q: Any tips for smaller development teams, say one or two developers and one tester?

A: If it's a smaller project, obviously you'd have fewer resources to devote than a large project, But at the same time, your user commitments would be smaller than on a large project, if your team is that small. So I would think (and this is just a guess on my part, I haven't tried it) that in a scaled-down fashion something like this would be applicable.

Q: Do you think if a more modern language, like C++, had been used to start with many of the original maint. issues might have been avoided?

A: A language like C++ might have helped, but it wouldn't by itself solve the problem. We have a saying at Los Alamos: "A determined programmer can write Fortran code in any language." It is entirely possible to write tangled code in C++; the Lakos book *(see slide 18)* has some examples of that. So even in C++ some of this kind of work would still need to be done.

Additional questions I didn't get to answer during the webinar:

Q: Did "cleanup" mean *only* dependency cleanup or more?

Q: Does "cleaner code" mean dependency clean up, or "expressive code", or both, or something else?

A: On our first pass through the process, the part where I was showing the dependency graphs, "cleanup" meant primarily dependency cleanup. Once that was done, we started to work on other kinds of cleanup as well: a bit of leftover dependency cleanup, as well as more expressive code, new physics methods, optimizations, and so on.

Q: It must have slowed some releases to spend all of this time on backend work. Were users supportive and patient?

A: For the most part, yes. One of the things we were careful to do, as this process started up, was to work with the users' management and make sure they understood what we were doing and why; and once we did that, they were generally supportive.

Q: Did the untangling process make the code longer or shorter?

[Listener comment: On the question "Did the untangling process make the code longer or shorter?" addressed to everyone, my take it that it may vary. In many cases it will lead to an easy to understand code and a shorter and modularized code.]

A: I don't know the exact numbers, but I know that some parts of the code ended up slightly longer. As part of the untangling, we were making some of the subroutine interfaces more explicit, and that made the declarations and calls longer *(see slide 14)*. But at the same time we were removing some old, unused code; so perhaps that balanced out some of the parts that got longer. And in any case, the changes bought us modularity and clearer code, and we thought that tradeoff was worth a small increase in code length.