by daniel chudnov
Information Technology Specialist
Office of Strategic Initiatives
Library of Congress

How to Rewrite Software—And When Not To

EVEN KNOWING THESE RULES, I BROKE ONE OF THEM. ONE YOU SHOULDN'T EVER BREAK: DON'T REWRITE YOUR CODE FROM SCRATCH.

Nobody ever accused me of being a great coder. I've picked up some skills over the years and have seen little decisions lead to big problems, so I've learned a lot of what an experienced programmer should know about the fundamentals. Do comment on anything nonobvious. Don't repeat yourself. Do use consistent names for classes, functions, and variables. Don't optimize before you have to. Do use version control. Don't generalize until you need to.

See how I talk a good game?

But it's a sham. I've made a mockery of these time-honored rules, and the result is a travesty.

Even knowing these rules, I broke one of them. An important one. One you shouldn't ever break. The one Joel Spolsky calls "the single worst strategic mistake that any software company can make" (www.joelonsoftware.com/articles/fog0000000069.html). That rule, which is so critical I'm devoting a column to telling you about how I broke it and the "traveshamockery" it left me with, is this:

Don't rewrite your code from scratch.

Don't Rewrite Your Code From Scratch

I started rewriting one of my applications from scratch, and it's ruining me. It's making me hate software development. It's making me question everything else I do. It's forcing me to question everything I thought I knew. It's frustrating me so much I don't want to talk about it. And here's the worst part:

It's not done.

Why not?

That's the first lesson behind the "don't rewrite from scratch" rule. Don't do it because it will take longer than you think—much longer. Long enough that it will make you put off other tasks you should be attending to, which you probably should've done prior to, or instead of, any rewriting at all.

In my case, I'm rewriting a nearly 6-year-old web application. I hadn't done much to improve it in the past 2 years—until a few months ago it had just been humming along nicely, with a manageable amount of duct tape and bailing wire, getting restarted every time it felt like stopping. Sadly, one of its pieces broke. The sane thing to do would have been to fix or replace that piece and get on with life. And that's the key thing to remember from this lesson. If your tech person says, "This part broke, so I'm going to rewrite the whole thing," don't let him or her do that. Make the person write up exactly what broke, how it broke, and what it will take to fix that one thing. Then, make your tech person write up a separate plan for rewriting the whole thing—no glossing over the details. Finally, compare both plans. Nine times out of 10, you can fix or replace the piece that broke and go happily back to other work.

What did I do when one of my code's parts broke? I started rewriting it from scratch. And I still haven't finished. I started over for bad reasons. The code was old. I wanted to upgrade

or replace several different components, in addition to fixing what broke. In the past, I'd used some components that were not well-known by many developers, so few people were able to help me when I got stuck, or at least fewer than there would have been if I'd used something more common. And there are programming frameworks that are more modern and do more heavy lifting for you. All that and I didn't even have to use the new tools as a beginner—I had already built a substantial application with it previously, and I knew it could be made to work. (For the coders among you, I'm talking about rewriting the unalog.com web application, which ran for years on Quixote-1.2, ZODB-3.2, and PyLucene-1.0, swapping in recent releases of Django, PostGreSQL, and Solr, thinking it wouldn't take very long.)

So I dove in. At first, I made a lot of progress quickly. But then, reality kicked in, and I hit the first of several walls.

Why Does It Take So Long?

As Spolsky wrote, old code, no matter how messy it might seem, still represents a lot of hard-won knowledge. The accumulated bug fixes, strange knobs and dials, and hard-to-understand logic that old code contains is worth a lot—it is the realization of knowledge needed to deliver your service to the real people who use it. Spolsky's piece is 9 years old, but it is full of details and advice about this exact concern, the sum of which is that when you start over, you have to re-implement all of that knowledge somehow.

That's the first strike against me.

Then there's the illusory promise of "new components" I was going on about before. A lot has changed in web development since 2003. Today's popular web frameworks do more things for you automatically that you had to do for yourself back then. For instance, Django simplifies common database access and session management tasks for you, both of which I had to implement for myself back then. When I got started with the rewrite, these tasks were easier to do than they were the first time, so I had the old data ported to the new database quickly. It felt like I'd made a good decision to rewrite, and I had good momentum to plunge all the way in.

That's where the illusion stopped. The tasks that followed were much harder than I anticipated. Next, I had to transpose all those hard-won tidbits of knowledge specific to my application over to the new environment. A few of these were easy, but when it got hard, I got frustrated. The problem was that those tidbits that were based on a bug fix here and a weird condition there were expressed in code that applied to the old components and frameworks. Some of them didn't transpose well because their old solutions fit those older pieces in a way that didn't fit the new ones. Sometimes I found a ready alternative approach with the new stuff. However, like when you move to a new home and there's a piece of furniture or a work of art that just doesn't have an obvious place to go, lots of old tidbits from the old code don't always fit nicely into the new code. Maybe it's just because people learned better than to do it that old way of yours, or maybe the new framework has its own odd approach that really doesn't let you translate your stuff well. However it happens, each one of these cases is another wall you hit, another reason to stop working for the day, and another obstacle that makes the whole endeavor seem less pleasant, diminishing your enthusiasm for the rewrite.

Which is two strikes against me.

Finally, and perhaps most importantly, because it explains my reaction to hitting these obstacles, I'm not the same person I was 6 years ago. I'm not writing something that doesn't exist yet. I'm not using it to get a paper published or to help win a grant. I'm not doing something new, and I'm not physically or professionally in the same place I was 6 years ago. When I ran into obstacles back then, I pressed on, anxious to deliver the application and to show off exciting new features. Now, I just miss my old application and wish I was done already. If this were you, maybe you'd have different staff, other priorities, or different budget constraints; maybe you'd have some of my issues too because that's what happens over time. You can't step in the same river twice, right?

But I'm me, and 4 months after taking on this rewrite, which I figured might take a month of nights and weekends ("Oh, a month, tops," as people like me are wont to say), I'm not done. I need another month or so to finish up those pesky tidbits before I can turn this on. A month, tops!

Three strikes. I'm out.

Exceptions to the Rule

I don't want you to think there are never good reasons to rewrite an application from scratch. Sometimes an old car just needs a tuneup, new brakes, or new tires. But, eventually, the engine goes, and it might make sense to buy a new car. But you can't do that with software! Somebody still will have to re-implement all that stuff. So when is it worth it?

If you have an application built around some data or software component that isn't formally supported anymore, you're on your own. And every new change touching that old bit of code might cost more time and money. You might be better off if you're using free/open source software—it might be easier to find others in the same situation to collaborate with, but it's not guaranteed that you will. You would at least have the option of doing something with the unsupported pieces, though—an option you probably wouldn't have with proprietary code. Free/open or not, though, the cost equation might be such that it just makes sense to start over.

Another case might be when you inherit incomprehensible code. It might be written in an ancient language or just written very strangely or very poorly—trust me, I've seen all three, and I've done all three of those myself. I pity the poor souls who've had to clean up my messes, and I

understand when somebody says they replaced my work if I know it was that bad. It might just be easier to document the working functionality of the old stuff and use that as input for a replacement.

Another acceptable reason can be when an application really is small enough to replace quickly. What's small enough is hard to say. Does its code fit on one screen? Is it no more than a few files and a few thousand lines? These are arbitrary questions, and there's no hard and fast rule. In my experience, if you think it will take a few hours, it'll take a day, and if you think it'll take a few days, it'll take a week, and so on. If your estimate is short and you come in under that time, good job. If you run over a short estimate, you might re-evaluate. If the estimate is long, you might consider a brief attempt at a rewrite as a learning exercise. You want to learn a new language? Re-implementing something you already know is a good test to get a feel for the new language. Your coder wants to switch frameworks? Let him or her try rewriting a piece of the old code with the new tools for a short period. That can give him or her a feel for the new option, and it will give you a better assessment of how long the whole thing might take. But don't forget that the first bits are often the easiest ones because the coder will always pick something to do first that's easier with the new stuff—or that's what I do, and what I did. The harder parts rarely get replaced first (but if they do, that can be a good sign, though what's hard might have just changed).

Maybe all programmers break rules like this one from time to time. That could explain the variety of books about programming as a science, as an art, and as a discipline to be practiced in one faddish style after another. And that might at least help me feel a little better. But that kind of reassurance isn't going to help me finish this rewrite. ■

Daniel Chudnov is a librarian working as an information technology specialist in the Office of Strategic Initiatives at the Library of Congress and a frequent speaker, writer, and consultant in the area of software and service innovation in libraries. Previously, he worked on the DSpace project at MIT Libraries and the jake metadata service at the Yale Medical Library. His email address is daniel.chudnov@gmail.com, and his blog is at http://onebiglibrary.net.