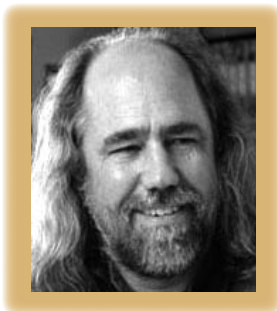


Nine Things You Can Do with Old Software

Grady Booch

Of course, “old” in software years is a subjective thing. Some are still running 20-year-old Cobol programs; others use even older assembly language programs. By any measure, those are classic examples of old software. However, I’ve also encountered organizations that are running the best and the brightest situational-Web2.0-agile-social-networking-enabled-*semantically-dense-cloud-based-systems-of-systems-script-based* software that, while only a few weeks old, is “old” to them, simply because they’ve moved on to the next best and brightest thing.



Now mind you, I’m not casting stones at either practice. Those organizations running truly aged software typically carry on for purely economic reasons: what they have is good enough. For those running fresh software, they tend to do the voodoo they do for one of two reasons. Perhaps their market demands extreme innovation. Or, while they’re engaged in this breathless dance of marketplace and technology, perhaps they haven’t yet converged on a stable architectural base upon which to grow their business. So, churn is actually desirable—if not inevitable.

However, I have a theory that organizations that maintain decades-old software often practice a false economy relative to the true costs of running that old software. Similarly, I believe that fresh organizations often underestimate the mass and therefore the attendant inertia and costs of what appear to them to be “mass-less” lines of code. Those in organizations with vintage software often grok these latter issues because they’ve felt the weight press down on them over the years, whereas those in younger organizations often don’t grok it, simply because mass isn’t yet a limiting factor. Dare Obassanjo, a program manager at Microsoft, put it this way (<http://web2journal.com/read/603827.htm>):

There is no legacy code at a startup. When your code base is young, it isn’t a big deal to have developers checking in new features after an overnight coding fit powered by caffeine and pizza. For the most part, the code base shouldn’t be large enough or interdependent enough for one change to cause issues. However, it is practically a law of software development that the older your code gets the more lines of code it accumulates and the more closely coupled your modules become. This means changing things in one part of the code can have adverse effects in another.

This poses a pragmatic question: whether you measure “old” by weeks or years, what can an organization possibly do with old software? One of nine things.

Abandon it

In other words, just throw it away. When you’ve exhausted a piece of software’s entire economic value, sometimes the best thing to do is to get rid of it. If you’re talking about scripts that live at the edge of a Web-centric system, or code that glues systems together, blowing a code base up and starting fresh makes clear sense because the cost of change is so low. Indeed, some degree of this scrap-and-rework is to be expected in every domain, especially those in extreme innovation mode (but it does get annoying if you never converge on some stable form).

Sometimes, you might abandon your code base not because you want to but because the marketplace has forced you to do so. When a company implodes in a messy fashion, its software often suffers collateral damage, because software without a cultural and tribal memory to attend to it is typically just so many bits in the ether. Occasionally, you can extract some value from the patents associated with that software—but that’s a topic for a future column.

Give it away

Placing useful software into the open source community is a viable choice, viable in the sense that this is one way an organization can give back to the industry and its domain. I've done this with a library of Ada components (and I'm astounded that groups are still using them). I had no desire to continue with them yet knew that someone could still extract value from them.

Ignore it

If your code base is good enough, meaning it's stable and does what you need it to do, then sometimes you'll continue to use it but never touch it. I've seen this in the small—applications that link in algorithmically intense components—as well as in the large—you'd be surprised how many banking systems contain such old chunks. Sometimes, you ignore your code base because it's really good enough. Sometimes you ignore it because the original developers are long gone and you're afraid to touch it, lest it crumble into dust at the slightest jiggle.

Put it on life support

Sometimes your software is good enough but the platform on which it runs is ailing. I've seen this happen with air traffic control systems and train-switching systems around the world: an organization has software that works just fine, but the hardware on which it runs is rapidly falling apart (and there are no replacements in sight). The situation might be compounded by the fact that you no longer have the original source code, so you can't even recompile it to a new platform. In such cases, I've seen organizations go to extreme measures, either buying the remains of old machines to cannibalize (which only delays the inevitable) or building platform emulators to run the original code unchanged.

Rewrite it

The time might come when you decide you just can't afford to keep an existing code base around: it's just too expensive to maintain or operate, and it's just too brittle to change. A somewhat reasonable reaction to this scenario is to send a team off to rewrite the system from scratch. At best, this brings a much-needed breath of fresh air to an otherwise musty code base. However, there be dragons here: although this path is tremendously appealing, duplicating an old piece

of software's complete operational behavior is almost impossible. You might be tempted to clean up obvious bugs in the code base, but you'll probably find that some users (humans or other systems) rely on semantics that they think are features. Therefore, if you go down this path, you must be prepared to either spend a lot of time nailing down the system's full formal operational semantics or deal with gently transitioning your user base.

Harvest from it

This scenario is similar to the previous one, except that this approach starts from the inside, not the outside. An old piece of software might be creaky and brittle, but it's always been inherently solid from years of real use. Study any chunk of software, and you'll find algorithms, patterns, and abstractions that transcend their current manifestation, things that have real economic value. By harvesting these components and then using them as a knowledge base upon which to develop a new one, you'll have preserved the experiences that have proven themselves over the years. By the way, this is why I'm so jazzed about the notion of patterns: they offer a vocabulary for reuse that transcends any particularly technological instantiation.

Wrap it up

The most common problem of good-enough software is integrating it with fresher stuff. A technological impedance mismatch often exists between older mainframe-centric software and newer systems. This is where technologies such as SOA (service-oriented architecture) come into play. Take your good-enough software, put a wrapper around it, then press on. From the inside, it's the same old stuff; from the outside, it plays well with the younger generation. The trick, of course, is getting the wrapper in the right place, punching holes in the right place, and making those holes the right size and shape.

Transform it

This is the hardest thing to do, because it requires a steady hand, a solid vision for the future, and an intentionality that's often missing in development organizations. In the small, this is simply refactoring, a solidly proven practice. In the large, this is architectural transformation. If you have a chunk of valuable software, software that's so central to your business that you can't

turn it off and therefore can't do to it any of the other things I've mentioned, your only hope of keeping it fresh is to continuously transform it. This requires a steady hand, because short-term delivery pressures will always oppose continuous transformation. This requires a steady vision, because someone must keep an eye on the horizon, leading the organization to where it needs to be in one, two, five, ten years, while the team madly paddles with their heads down to the next release. This requires intentionality, because it requires investment that pays off in the long term, not the short, and thus is subject to the whims of the market and short-sighted managers. In practice, I've encountered exactly three organizations who put real economic weight behind intentional continuous transformation—and each one is a very successful business.

Preserve it

Sometimes the economic value of a chunk of software is completely spent, but its historical value is not. Being able to look back on the earliest compilers and operating systems or perhaps even games such as Doom tells us a great deal about the culture at the time. It would be a shame if future generations didn't have the source code to programs that have changed the world—and software has changed the world. Many museums preserve tangible objects; the Computer History Museum is in the business of preserving software. So, if you have bits of software that are historically interesting, please consider donating them. (If you deliver it to me in bits, I promise to pay the shipping costs.)

Economically interesting software represents a significant capital investment: there is a cost to creating, deploying, operating, and even owning such software (but also an attendant value). I therefore distinguish between preservation and evolution: we *preserve* old software for irrational reasons (such as intense aversion to risk); we choose to *evolve* old software when doing so contributes to its long-term value. So, in the end, economic reasons—not technical ones—drive what we should do with old software. ☞

Grady Booch is an IBM Fellow and one of the UML's original authors. He also developed the Booch method of software development (*Object-Oriented Analysis and Design*, Addison-Wesley, 1993). He's working on a handbook of architectural patterns, available at www.booch.com/architecture. Contact him at architecture@booch.com.