Editor: Diomidis Spinellis ■ Athens University of Economics and Business ■ dds@aueb.gr

Silver Bullets and Other Mysteries

Diomidis Spinellis

It seemed like a good idea at the time. —Ken Thomson, on naming the Unix system call to create a file "creat"

When conference participants interrupt a speaker with applause, you know the speaker has struck a chord. This happened when Alan Davis, past editor in chief of *IEEE Software*, gave a talk on improving the requirements engineering process at the NASSCOM (Indian National Association of Software and Services Companies) Quality Summit in Bangalore in September 2006. He was explaining why a marketing team will often agree with developers on additional features and a compressed delivery schedule that both sides know to be unrealistic. The truth is that this places the two parties in a Machiavellian win-win situation. When the product's delivery is inevitably delayed, the developers will claim that they said from the beginning that they couldn't meet the schedule but that marketing insisted on it. The marketing people also end up with a convenient scapegoat. If the product launch is a flop, they can say they missed a critical marketing time window owing to the product's delay. Where else are we playing such games?

Aging systems

Consider a 15-year-old software system. Its design doesn't match the environment it operates in, its original developers have matured during its lifetime, and hundreds of fixes and improvements have accumulated thick layers of "cruft" (redundant or poorly designed areas) all over its code base. Any sensible software engineer would argue that the system is ready for scrapping and rebuilding from scratch.

However, pointing out this fact is bad for all parties involved. It shows that developers haven't really done a stellar job over the years; they'll have to admit that many of their design decisions turned out to be incorrect. Getting a system's design wrong is natural because, first of all, the environment a system operates in changes the moment the system is installed and, second, because the expectations people have of a system change with time. To get a feeling of changed expectations, try typing a page on a typewriter—once considered to be the perfect tool for writing neat documents. In this context, hindsight is treacherous and unfair because it changes the rules of the game, after the game has finished. Although in the year 2000 Pets.com managed to raise US$82.5 million in an IPO, seven years later those same people who bought Pets.com stock would ask: "A site selling pet food over the Web? What were they thinking?"

Additionally, in our relatively young profession, many aging systems were originally writ-

 0740-7459/07/$25.00 © 2007 IEEE

ten by rookie programmers who were cutting their teeth on code for the first time. So, the product is likely badly designed, its code lacking in structure and consistency. I often look at code I wrote several years ago, and I can immediately realize in which phase of programming immaturity and folly I was in. There was a phase when I thought that "shrt idntfrs wr cool," one where I tried to exploit every trick of the C programming language—because I could—and one where I hadn't yet learned to comment my code (even if the only eyes that would see it were my own). I wonder what I'll think in 10 years of the code I write now.

Even if a system were perfectly designed to match its environment, 10 years later, its code would still show the signs of time. Successive fixes and improvements typically violate initial assumptions. Developers who fail to understand an aspect of the system's design will add their bit in a different way. Or, even if they understand the design, they might not understand the system's coding conventions and use different ones. This occurs often in systems written in languages such as C++, where incompatible identifier naming conventions coexist, even within the language's own libraries. Worse, other developers will duplicate code, violating the Don't Repeat Yourself (DRY), single point of control principle, increasing the risks of future changes. In sum, the code will accumulate cruft and become unmaintainable.

We're accustomed to aging in the physical world. We know that people, dogs, cars, ships, clothes, and computers have a finite lifetime. Our experience with immaterial creations is mixed. The works of Homer, Shakespeare, Mozart, and, dare I say, the Beatles haven't really deteriorated over the years. On the other hand, a journal article in software engineering passes its prime (it reaches the so-called aggregate cited half-life) in eight years; in the sprightly field of nanotechnology, this figure is just four years. Unfortunately, we haven't yet come to terms with the idea that software ages, often beyond salvation.

On to silver bullets

So what can developers do when faced with an aged software system? They could simply come clean. Claim that the code that they were paid to design, write, and maintain is a pile of excrement and ask for another chance to do it right. Even the most thick-skinned and politically naive developer will, however, realize that this isn't a smart move. Coming clean is also a problem for the developers' managers, because they'll have to explain the mess to their higher-ups.

This is where a silver bullet comes in handy. Imagine a system that cost $300,000 to develop and in which, over the years, its owners invested another $700,000 to maintain and enhance. Starry-eyed managers might think they have a system worth a million dollars on their hands, but we all know that due to its age, the system's real value is a tiny fraction of that. The developers continually find themselves in the uncomfortable position of having to explain why new changes are so costly and time-consuming and why each improvement and fix introduces so many new bugs.

One day, a godsend order for a major enhancement comes in, and the developers estimate its cost at $500,000. Before their client has time to recover, they claim that a revolutionary new technology is available that will let them build with this sum both the existing system and the new enhancement from scratch. Moreover, by adopting this new technology, future enhancements will cost only a fraction of what they would cost using the old technology.

The precise nature of this technology claiming to offer dramatic productivity improvements is unimportant. At various times, this silver bullet has been known by names such as structured programming, object-oriented languages, 4GLs (fourth-generation programming languages), CASE (computer-aided software engineering) tools, RDBMSs (relational database management systems), XML, visual programming, *n*-tier architectures, managed code—the list goes on. What's important is that the move suits everybody perfectly. Developers can abandon their old code without having to explain the awkward truth; they'll also get to update their technical skills and brighten their employment prospects. Managers will be seen as heroes for taking a bold step with the new technology (in management, action is often mistaken for achievement). Conveniently, at this point an army of vendors will also step in to offer to their eager listeners supporting evidence and success stories. And—this is the icing on the cake—by following this route, developers and managers also buy an insurance policy. If the transition plan fails and the bullet's magical productivity increases don't materialize, they can claim that the technology is still immature or had hidden flaws.

What's my opinion of this charade? Software ages and becomes increasingly expensive to maintain. New technologies offer modest but not spectacular improvements in productivity. It's therefore sensible from time to time to rebuild a system from scratch. It might be harmless and politically expedient to claim that we've found a silver bullet, but it's even better to know what we're really doing.

We haven't yet come to terms with the idea that software ages, often beyond salvation.

Diomidis Spinellis is an associate professor in the Department of Management Science and Technology at the Athens University of Economics and Business and the author of *Code Quality: The Open Source Perspective* (Addison-Wesley, 2006). Contact him at dds@aueb.gr.

Post your comments online by visiting the column's blog: **www.spinellis.gr/tools**