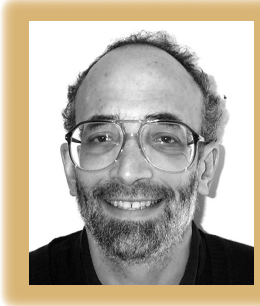
Contact Editor: Dale Strok ■ dstrok@computer.org

The Software Engineering Silver Bullet Conundrum

Daniel M. Berry, *University of Waterloo*

In 1986, in his famous "No Silver Bullet" paper,[1] Fred Brooks predicted, on the basis of his experience, that

There is no single development, in either technology or in management technique, that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity.



That is, he predicted that in the next 10 years no software development silver bullet would be found. (A silver bullet is the only kind of bullet that will kill a werewolf and thus solve the problem of its terrorizing the countryside.) In arguing for his claim, he added,

I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation. We still make syntax errors, to be sure; but they are fuzz compared with the conceptual errors in most systems.

If this is true, building software will always be hard. There is inherently no silver bullet.

The conceptual errors he was talking about involved failing to capture the essence of the system being built—that is, the system's conceptual construct, the system's requirements.

Brooks divides software system concerns into the essence and the accidents. A system's essence is, as I just mentioned, its requirements and what it does; its accidents are the technology used to construct it. The technology is termed "accidental" because the choice of, say, programming language has a much smaller effect on the difficulty of building a system than the system requirements do.

Although we have yet to find a surefire way to understand a system's requirements, we have over the years made significant technological improvements that have combined to increase software productivity by more than an order of magnitude. These improvements include high-level language compilers, configuration managers, testing tools and harnesses, debugging tools, and GUI builders. In other words, we've found or devised many technological aluminum bullets that have combined to be almost a silver bullet, while no bullet itself is silver.

The pain of development

In 2002, I went further and claimed that there would never be a silver bullet unless a technology could deal with not only the essence of software systems and their requirements but also the relentless changes to these requirements. In "The Inevitable Pain of Software Development," I argued that the typical software development method is effective in its first application to any system development problem.[2] However, once developers have built and deployed a version with the method, the requirements begin to change, whether from E-type system pressures[3] or client and user demand. When an inevitable change comes along, modifying the method's documenting artifacts is so painful that developers avoid doing it the right way, by carefully tracking a change's effects. Instead, they create a quick patch that increases the system's brittleness.[4]

 0740-7459/08/$25.00 © 2008 IEEE

Since writing that paper, I've realized that there are two more fundamental reasons why we'll never have a silver bullet.

First, a silver bullet kills itself and ceases to be a silver bullet simply because once we have it, we'll quickly solve all the formerly too-tough problems that the silver bullet lets us solve. Doing this brings us to a new frontier of not-easily-solved problems. Then, owing to our very human ambition to advance, we try to solve problems that are just beyond the frontier; the silver bullet has become an ordinary lead bullet with respect to these problems. In other words, as Krzysztof Czarnecki noted in a private communication, it's as though after we use one silver bullet to kill one werewolf, all werewolves adapt and become immune to silver bullets.

Second, the cause of the inevitable pain is the very act of writing something formal so that it will be implemented. Once we've written that formal specification, executable or not, we're stuck. Even if we don't write anything traditionally called a formal specification, we do eventually write executable code, which is a formal specification. Any subsequent change in requirements requires changing the specification in a way that preserves correctness. So, we have pain. Repeatedly changing a specification is painful, redoing a specification from scratch is painful, and deciding which of the two pains to endure is painful. This pain happens even if we use a silver bullet, which is about to convert itself into a lead bullet. The only way to avoid the pain is to not write any specification. However, then we get no implementation, unless we build a machine that reads our minds, intelligently fills in all the details, and does what I mean (DWIM), an impossibility.

Always a new target

In a private communication, Diomidis Spinellis offered another way to view the first reason:

> *As soon as a particular application domain becomes easy (in effect, solved through a silver bullet) we move on (partly thanks to the relentless hardware advances) to more difficult problems, for which, by definition, there are no silver bullets.*

A concrete example of this is GUI builders. Before such things existed, few people built systems that required high-resolution, interactive graphics because they were just too hard to program. We simply didn't try to build such systems, and we didn't require such systems to be built. Instead, system requirements spoke of minimally interactive, command-line interfaces.

One fine day, X Windows[5] appeared with its platform-independent library of easily invoked and used widgets. (Actually, the Macintosh user interface predated X Windows. The Macintosh user-interface library was available to Macintosh software implementers, but X Windows was the first widely available platform-independent GUI-building library.) Overnight, it became easy to build applications with interactive, high-resolution GUIs with standard looks and feels. Then, people began to conceive and build all those previously inconceivable systems that demanded such interfaces and had therefore been completely ignored.

There are many other examples of such technologies—for example, relational database management systems, Web servers, Wikis, lexical analyzers, parsers, string manipulators, cryptographic systems, and digital typesetters. Each of these systems solves what once was a hard problem but has become, as Spinellis observed, "a tool or an API away from us."

Nowadays, such applications are as routine as compilers and other largely manufactured applications. Today's impossible interfaces involve sounds and odors; we simply ignore systems requiring these interfaces, considering them impossible to build.

No bullet can be silver for more than an instant. That each silver bullet quickly becomes an ordinary lead bullet is the basic conundrum of software engineering silver bullets. Does this conclusion mean that we should stop trying to improve software engineering? No! However, we need to stop the search for silver bullets and to focus on finding aluminum bullets. That an aluminum bullet is lighter weight than a silver bullet of the same caliber is a deliberate part of my point—software engineering methods must be lightweight. Moreover, we need to stop pouncing on each good bullet that we do find and hyping it as a silver bullet that can solve more problems than it actually can.

References

1. F.P. Brooks Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, vol. 20, no. 4, 1987, pp. 10–19; originally published in *Proc. IFIP 10th World Computer Congress*, North-Holland, 1986.
2. D.M. Berry, "The Inevitable Pain of Software Development: Why There Is No Silver Bullet," *Radical Innovation of Software and Systems Eng. in the Future, Proc. 2002 Monterey Conf.*, LNCS 2941, Springer, 2004, pp. 50–74, http://se.uwaterloo.ca/~dberry/inevitable.pain.html.
3. M.M. Lehman, "Programs, Life Cycles, and Laws of Software Evolution," *Proc. IEEE*, vol. 68, no. 9, 1980, pp. 1060–1076.
4. L.A. Belady and M.M. Lehman, "A Model of Large Program Development," *IBM Systems J.*, vol. 15, no. 3, 1976, pp. 225–252.
5. R.W. Scheifler and J. Gettys, "The X Window System," *ACM Trans. Computer Graphics*, vol. 5, no. 2, 1986, pp. 110–141.

Daniel M. Berry is a professor in the Cheriton School of Computer Science at the University of Waterloo. Contact him at dberry@uwaterloo.ca.

No bullet can be silver for more than an instant.