One Giant Step Backward

Old saying: "The more things change, the more they remain the same" … or do they?

It is common for software developers to talk about the rapid change of pace in our field. We say things like "It's hard to keep up with all the things that are happening," and make apologies for not being up to date on the latest whatever. I've been guilty of that myself, on occasion.

But do you know what? I don't really believe it. Almost not a word of it. Except for various vendor products, such as tools and processes, the things I learned in software kindergarten, way back in the 1950s, are for the most part still valid today.

Computer hardware developers, I would assert, have made great strides in moving their field forward over the years. Smaller/faster/cheaper is almost a mantra of their fast-paced field. We software folk, by contrast, tend to build software in the same old ways. Reuse? Libraries were common in the 1950s. Coupling/cohesion and information hiding? We knew about those back then, too, although we didn't call them that. Programming languages and operating systems? The field didn't begin with them, but by the late 1950s (way back in *those* dark ages!) they were very common.

I realize you may disagree with me on this. There are lots of things that have come along since then to help software developers, such as methodologies and tools and processes of various kinds. But I would assert that many of them are pimples on a blister, refinements on a theme, not that significant in the overall history of the software field. Let your letters of response fly!

The point I want to make in this column, however, goes even further than my "we ain't made no progress" cry: I believe we've actually lost ground in some areas. That is, I am suggesting the sacrilege that we actually did some things better back in those early days than we do today.

Case in point: programming languages. I want to make the case that, over the years, the programming language community has taken several steps backward. That's not to say, of course, that it has not made progress. Today's programming languages, for the most part, are significant improvements over those of yester-decade and yester-millennium.

But there is one important way in which programming languages have fallen backward—that way is application domain focus. Today's languages seem to ignore, almost totally, defining the application domain for which they are most appropriate. In fact, the more you look at today's languages, the more you realize they are intended to serve whatever application domain you happen to be working in. It's that "one-size-fits-all" phe-

JASON SCHNEIDER

I am suggesting the sacrilege that we actually did some things better back in those early days than we do today.

nomenon I've discussed frequently regarding other topic areas, such as methodologies, all over again, this time applied to the programming language field. I've seen conferences on "domain-focused programming languages," but I've never seen anything of widespread significance to the field emerge from them. In fact, computer science, over the years, has fragmented into such solution-focused specialty areas, that I am not convinced that one subdivision of the field (including the ones looking at domain-focused languages) communicates all that successfully with all the others. Ah, but that's a topic for another column sometime!

Let me get specific here. When software folks first invented higher-level programming languages, we did it with a particular application domain in mind. Fortran for scientific/engineering applications. Cobol for business applications. RPG for Report Generation. A variety of languages for system programming, including one called SYMPL (systems programming language). A different variety of languages for real-time programming, including CORAL and JOVIAL and, eventually, Ada.

Most of that language progress dates back to the late 1950s. We were just beginning to understand the diversity of applications that software could be put to and we focused very carefully on the domains whose problems needed solving. It wasn't just a language thing, of course. *Communications*, for example, had various sections back then devoted to specific application domains. The whole field looked at the problems it had to solve first, and only after that looked at approaches for solving those problems.

What happened to change all of that? Perhaps the first and most obvious thing that came along was the programming language PL/1. IBM, seeking to find generic approaches to software problem solving, decided to produce a programming language that combined all of the facilities of the then-best-known and application-specific languages Fortran, Cobol, and Algol: PL/1 was the result of that effort. Many applauded PL/1, noting it was part of a broader trend toward application-independent computing approaches (computer hardware, in the early days of the field, was also problem-domain focused, and IBM's 360 architecture, like its PL/1, was an attempt to produce one product line to serve all of IBM's customers).

But not everyone thought the 360 and PL/1 were such good ideas. Folks made fun of PL/1 as "the kitchen sink language," meaning it contained all the imaginable language features, except, perhaps, the proverbial kitchen sink. There was enormous resistance to PL/1 (although not, interestingly enough, to the 360). Eventually, PL/1 was largely a failure. Fortran and Cobol rolled on (as PL/1 began quietly dying), surviving—of course—to this day.

Perhaps it was an accident of history, or perhaps it was not. But about the same time the 360 and PL/1 appeared on the computing practitioner scene in the mid-late 1960s, the academic field of Computer Science (CS) also was born. And CS, for better or for worse, embraced domain-independence wholeheartedly. It may not have liked PL/1 all that well, but the languages CS did like—the various flavors of Algol—were considerably more problem independent than the Fortrans and Cobols of their time.

One of the favorite topics of academic CS was, and to some extent still is, programming languages. And, language by language, the CS-invented languages lost all flavor of domain-dependence, and became generic in their approach.

To some extent, that has been a good thing. As the breadth of software applications has spread so dramatically, it would have been impractical to invent yet another language for yet another domain. And yet, I would assert, throwing out the domain-focused babies like

Fortran and Cobol with the domain-independent bathwater has not necessarily done the field any favors. I have lamented the purported demise of Cobol elsewhere, not so much because it is a good language for business applications (although it still is!), but because no one bothered to invent a better one before deciding that Cobol needed to be discarded.

There was an interesting reprise of the domain vs. generic programming language battle in the 1970s, one that extended for a decade into the 1980s. The U.S. Department of Defense, for a variety of good and not-so-good reasons, decided it needed a new programming language to serve real-time applications, especially those built for the Air Force, the Navy, and the Army. And it proceeded to perform the most careful domain-specific language study of all time. It gathered requirements for such a language from the tri-services and from tri-service vendors, and scrubbed them over and over again, and held a language competition, and eventually picked and implemented a language. A real-time language, it is important to reiterate. Ada, like PL/1, was not universally liked, but it was very un-PL/1-like in one respect—it was domain specific, just as PL/1 had been domain independent. At this point in time, at least, the last of the domain-focused languages!

And then something odd happened. The DoD had trouble getting its vendors to use the language, in spite of a variety of proclamations requiring that it be used for most DoD real-time applications. As time passed, the DoD—desperate to increase usage of the language that had cost it so much to develop—decided to broaden its domain of applicability. Originally focused on the real-time domain, with some support for systems programming so that its compilers could also be written in the language, Ada began to be seen by the DoD as a business-domain-focused language! Conferences were held on revising the language to include business-focused capabilities. Language modifications were defined. Articles advocating Ada for all application domains began to appear in computing publications. It was apparent that Ada, caught in its real-time death throes, was the subject of an emergency salvage operation, one designed to make a domain-independent sow's ear out of a domain-specific silk purse.

Mercifully, all of that domain-independent repositioning came to naught, and Ada died the kind of slow death that PL/1 has. It is interesting that the life-cycle stories of the two languages are so similar, given their domain-related nature was so different.

Let's do a reprise at this point. Domain-specific languages were once prevalent in the software realm. They have faded, but not disappeared, as domain-independent languages have taken their place. The various flavors of C, including most recently Java and C#, make no bones about being applicable to a wide variety of domains, and offer few if any of the features that business or scientific programmers, for example, need. There are, of course, GUI languages, such as the "Visual" languages, which are indeed focused on a particular domain—user interfaces. Even these languages, it is interesting to note, are as much about a particular aspect of the solution approach (graphical user interfaces) as they are about the basic problem domain.

So where does the programming language field stand today? The best-known languages are domain independent. The domain-specific languages are derided, but still used. The new languages that come along—and they appear much less frequently than they once did—seem to follow in the domain-independent footsteps of their predecessors. Nothing appears to be happening to change that.

What would I like to see happening in this same world? Research devoted to specific domains, and their specific needs. Research devoted to languages that meet those needs. The development of newer, domain-focused replacements, for those tired old specialty languages Fortran and Cobol. And an acknowledgment that, way back in the 1950s, we knew something we subsequently lost along the way. That domain differences matter. C

Robert L. Glass (rlglass@acm.org) is the publisher/editor of the *The Software Practitioner* newsletter and editor emeritus of the *Journal of Systems and Software.*