

On Our Failure to Embrace Advances in Software Technology

To what extent does the practice of software development take advantage of advances in software technology of the sort that have the potential to drastically affect the cost or quality of the software we build? Such advances occur infrequently, but over time they accumulate, often providing us with radical new and powerful approaches.

My answer to this question is “far less than it should”. While I am aware of no sound statistical basis for this answer, observation of practice has convinced me that the potential of many advances is not being realized. In fact, my conclusion is that we are failing to embrace advances in software technology to a significant degree. The impact of this state of affairs is serious, and it needs to be addressed.

Those processes known as *agile methods* are an example of an advance in software technology. Although they do not fit all development situations, in many circumstances agile methods can provide real technical value. As an advance, their contribution does not lie in cost savings or shorter times to market (oft cited advantages). Rather, it lies in their having challenged traditional stove-piped processes; in effect, they broke the mold. The demand for software continues to be significant, yet, despite their demonstrated capabilities, agile methods (along with many other innovations) have not been widely adopted by the majority of developers whom they could help. As a result, an opportunity to address the challenges we face more effectively is being missed.

Despite their lack of adoption, agile methods must be considered a success story compared with modern, strongly typed languages. Vast amounts of software, including safety-critical software, is written in C. C is a language that is basically typeless and which has a confusing syntax. In practice, C has true utility only in rare circumstances despite its very widespread use. There are vastly superior alternatives that support a reasonable type system, a consistent and clear syntax, and which have high quality compilers. Ada is the obvious example.

Why is *typeless* vs. *strongly typed* so important? Strong typing forces the programmer to think more carefully and provides a technical basis for automatic analysis. Many common defects made can be found easily and reliably at compile time for software written in strongly typed languages. Without a decent type system to work with and with an almost unconstrained use of pointers, C compilers can do little to protect the programmer from himself or herself.

But it is not just strong typing that modern programming languages bring. Ada has a comprehensive model of separate compilation allowing type checking across separately compiled units. It also has a semantic definition that permits explicit definition of what a program requires of the planned target thereby facilitating portability across platforms. A compiler for a new platform can analyze what a program expects in terms of things like numeric precision, and, if a carefully written Ada program compiles on a new platform, it will very probably run as expected.

These and other properties of modern languages provide the software engineer with a marvelous set of tools that can make his or her job much easier (but not easy). Programming languages are not the only tool that software engineers use nor is the creation of source programs the place where most software defects are introduced or costs are incurred. Nevertheless, creation of source code is both error-prone and expensive. One would think, therefore, that any engineer would choose the best possible tools. Yet the adoption of Ada and similar languages is essentially negligible compared to the adoption of C and C++, and this is true even for safety-critical systems.

Quite unexpectedly, even those development situations that are governed by some form of regulation or certification are not required to use modern programming languages. RTCA DO-178B, for example, is the software development standard required by the FAA for software to be used in commercial aircraft systems. If you have flown in any modern aircraft, DO-178B was the standard that the manufacturer had to meet for the on-board software, the digital fly-by-wire system on the Boeing 777, for example. Even for flight-critical systems, the DO-178B standard does not require the use of modern programming languages despite the fact that the standard was published in 1992, long after the initial Ada standard had been published. Even more unexpectedly, it seems likely that the replacement for DO-178B that is currently under development (to be known as DO-178C) will not require the use of modern programming languages either.

This strange situation deserves some serious thinking. To the best of my knowledge, failure to adopt new techniques that have significant advantages and low risk of adoption does not occur to any large extent in other engineering fields. What makes software engineering so different?

Actually, software engineering isn't different. The problem arises for many reasons, but one of the most significant is a variety of failures in higher education. Most professionals receive almost all of their formal training in colleges and universities. This places a tremendous responsibility on the shoulders of higher education, but, in terms of educating engineers about software, most curricula are not designed to meet this responsibility.

Modern engineered systems are software intensive to a remarkable degree, and developing dependable software for such systems in a cost-effective way is a very difficult intellectual challenge. Meeting this challenge requires two things: (1) that the engineers who build the software be properly trained; and (2) that engineers who are not properly trained not build the software. Both require explicit actions by the higher education system. The first requires that degrees in Computer Science and Computer Engineering include comprehensive coverage of the fundamental principles of software engineering. In many cases, graduates of these programs will end up developing software as professionals, and they must be able to meet the responsibilities they incur as a result. The second requires that degree programs in the other engineering disciplines include comprehensive coverage of the role of software in engineered systems, its complexity, and the potential dangers of building software without applying all the right techniques. Just as a software engineer would not design and build a load-bearing bridge based on an informal, ad hoc knowledge of civil engineering, so a civil engineer must not attempt to

build a significant software system based on a limited or informal, ad hoc knowledge of software engineering.

Curricula in all engineering disciplines tend to focus on a broad range of topics. The argument given for the current curriculum structure is that bachelors degrees are meant to provide a broad education in the principles of the discipline and are not professional training. In general, this is a laudable and defensible goal, and it is the right one for students headed to graduate programs. And it would be a satisfactory goal for those heading for professional positions if they and those hiring them knew that it was the goal and the practice. Most of them do not. Most employers assume that they are hiring graduates who know many of the latest advances and how the advances affect the software lifecycle. Most employers also confuse particular skills with principles. Employers demand knowledge of specific programming languages, for example, without realizing that understanding the principles of programming is far more important than knowing the syntax of any one language. A properly educated graduate can adapt to a new syntax very easily—in short, it is not the syntax of Java classes, it is the principle of information hiding.

Failure to adopt important advances in software engineering, advances that could reduce costs and improve quality, is imposing an unnecessary burden on society. Since higher education is at least partially to blame, it essential that we take a very close look at higher education, and the core of the solution lies in revisions to university engineering curricula. First and foremost, the time has come to abandon the idea that the education needed by an engineer who will become professionally involved in developing complex software systems can be obtained in a single undergraduate course entitled “Software Engineering”. It can’t. There is too much to learn and too little time in a standard four-year program. Electrical engineering is not a single course in a Physics program.

Next, Computer Science and Computer Engineering curricula need to focus on crucial principles. We can begin this process by revising introductory courses that presently focus on the syntax of Java or C++. They do little more than turn people away from the discipline. Introductory courses can take many different approaches and still remain useful, but their role should be to begin the process of educating the student in both the fundamental principles of the discipline and in the crucial notion of what it means to be an engineer.

Introductory courses should be followed by courses that continue the process of introducing and explaining the fundamentals of software engineering, showing how they should be applied, and illustrating them with the best possible tools including, agile development methods and other modern process concepts, requirements engineering techniques, rigorous specification, object-oriented and other design techniques, modern programming languages, inspections, metrics, static analysis, light- and heavy-weight formal techniques, systematic testing and associated coverage measures, and so on.

To complement these changes, degree programs in other engineering disciplines need to develop approaches to ensuring that their graduates have a suitable understanding of how difficult software is to build, its role in engineered systems, and the consequences of its

failure. Software has no obvious physical presence and software failure when undertaking a class assignment has little penalty. This gives engineers in fields outside of Computer Science and Computer Engineering entirely the wrong impression of the serious nature of software development.

This is strong medicine, but we have to take it—there is too much at stake to allow the status quo to continue. Some countries, most notably China, have already established national policies on software education. To summarize the challenge we face in one sentence, I can do no better than quote Peter Amey of Praxis High Integrity Systems: “We must decide we want to be engineers not blacksmiths.”