Continuing Prospects for an Engineering Discipline of Software

Mary Shaw, Carnegie Mellon University Institute for Software Research

In the late 1980s I helped establish Carnegie Mellon University’s Software Engineering Institute, which dramatically changed my appreciation of the critical problems in software development. Computer scientists had been talking about software engineering for two decades or so, and I wanted to better understand what it might mean to have an engineering discipline for software. This led me to look into the history of engineering, and the article “Prospects for an Engineering Discipline of Software” was one result.

Engineering fields typically evolve from craft practice of a technology, sufficient for local or ad hoc use. When the technology becomes economically significant, it requires stable production techniques and management control. The resulting commercial market is based on experience rather than deep understanding of the technology. Production problems often stimulate a related science, and an engineering profession emerges when that science becomes sufficiently mature to support purposeful practice and design evolution with predictable outcomes. Figure 1 shows this evolutionary path.

In “Prospects,” I concluded that software practice was sometimes craft and sometimes stable commercial practice. Some science was beginning to emerge, and isolated cases of engineering practice could be found—but that wasn’t the common case. Now, IEEE Software has offered me the opportunity to take stock again, with the field another 20 years older. Where are we now, and what are our prospects?

Programming Progress

In “Prospects,” I identified three eras in software research:

- **programming-any-which-way** (roughly 1960 ± 5 years), in which we wrote small programs without systematic understanding of even elementary abstractions;
- **programming-in-the-small** (1970 ± 5 years), in which we learned to apply systematic understanding of algorithms, data types, and formal reasoning; and
- **programming-in-the-large** (1980 ± 5 years),
in which we developed models for software structures larger than the individual program.

In the years since “Prospects,” the level of abstraction of programming language primitives has increased. Not only do general-purpose languages include, for example, object abstractions, but domain-specific languages incorporate powerful, but specialized, primitives.

Twenty years ago, object-oriented (OO) programming began to displace conventional procedural programming. Objects provide abstractions that match many applications better than hierarchical procedures; we now have a profusion of environments, frameworks, and design notations. These tools make OO programming an attractive alternative to classic procedural programming. Professional programmers now use these tools but adopt the object paradigm only to varying degrees.

Unfortunately, the complexity of the frameworks and tools is often daunting, and architectural decisions are often defaulted to the architecture implicit in the OO environments. I hope that eventually researchers can discover underlying simplicity that will vanquish the current complexity of OO development so that software developers can again turn their attention to choosing an architecture that fits the problem at hand.

Specific domains, improved understanding of the domain has led to models that raise the level of programming to the problem domain itself, with generators supporting automatic code creation.

As good theories and domain models emerge to support higher-level programming abstractions, they help move software development from the commercial realm to the engineering realm.

**Beyond Programming**

Computing technology has changed dramatically since 1990, thanks not only to Moore’s law but also to the public Internet. The resulting disintermediation of computing now lets people who aren’t computing professionals exercise direct control over their computational activities. Accordingly, the scope of an engineering discipline of software has expanded beyond large-scale bespoke software to include support for software widely used—and created—by the public. Let’s look at some of those changes.

Whereas the research community of 1990 focused on programs, we now see software development issues at a much higher abstraction level, with emphasis on marshalling and coordinating diverse resources and engaging end users in the software. I therefore mark more recent eras with higher-level ideas rather than programming ideas.

**1990 ± 5: Abstract Architectures**

In the mid-1980s, software researchers started recognizing the importance of explicit decisions about overall system organization. An early step was recognizing the varieties of abstract architectures that guide software system design. This led to classifying different kinds of components, the abstract protocols (connectors) that connect those components, and the architectural styles that guide consistent use of compatible components and connectors. Architecture description languages emerged to support these styles, and researchers began matching classes of problems with architectures appropriate for their solution. Companies began to consider architecture explicitly, and a project’s lead technical expert is now often called the “architect.” However, the connection between architectural styles and the code that implements them is still tenuous; both procedural and OO programming practices are based on the call-return relation between components, which largely fails to capture the useful abstract relations.

**2000 ± 5: Democratization of the Internet**

The most remarkable technical change of the last two decades has been the rise of the Internet as a principal engine of information and commerce. This has fundamentally changed the relationship between people and their information, communication, and computation.

Online applications, together with affordable PCs, have allowed everyday people to become not only consumers but also producers of content. The resulting disintermediation of software development has increased the number of people developing software, tailoring their own applications (www.pewinternet.org), or creating active Web content.
by two orders of magnitude, with professional programmers now a small minority.6,7

2010 ± 5: Vanishing System Boundaries
More speculatively, I see challenges emerging from the deep interdependence between very complex systems and their users. Problems facing software engineers are increasingly situated in complex social contexts, and delineating the problems’ boundaries is increasingly difficult.8 The Internet, for example, isn’t simply a communication and search infrastructure—its richness and vitality arise from the innovative applications, extensive content, and direct participation of its users, including e-commerce, social networks, entertainment, reference information, and—yes—spam, viruses, and phishing.

The Internet is an example of an “ultra-large-scale” system in which there’s no possibility of central control, no clean system boundary, and no single requirement to define correctness, except in the narrow sense that the infrastructure behaves correctly. Various independent, competitive, and creative users drive its evolution, and the challenge of keeping such a system under control goes beyond technical software considerations into social, economic, and political issues. The public-planning community recognizes this class of problems as “wicked.”9 Like it or not, our engineering discipline must interact effectively with these interests.

But Are They Engineering?
Of the approaches I’ve mentioned, software architecture is closest to the merger of science and established practice that supports engineering. Although good models and theories support the Internet’s infrastructure, they’re only beginning to emerge for the ecosystem of applications and constituencies. As for wicked problems, we’re just beginning to understand the problems and extent to which engineering approaches can contribute to their solution.

Management of the Software Production Process
Twenty years ago, the Software Engineering Institute’s CMM focused attention on the need for good software project management. The high-ceremony processes this model stimulated were soon challenged by agile and open source approaches that emphasized technical activities and low-ceremony management. By drawing attention to the management of software processes, the CMM helped software development move from craft to production practice (see Figure 1).

It’s unfortunate, though, that this took place—and continues to take place—under the label of “software engineering.” Software has engineering challenges aplenty, and mislabeling management and process issues as “engineering” diverts attention from the equally important technical issues of creating a systematic, scientific basis for an engineering discipline. Our prospects would be better if we’d recognize the former as “software management,” allowing the latter to fully occupy the mindscape of “software engineering.” No other engineering discipline suffers this confusion. As for our field’s maturity, whenever someone says, “We don’t need technical advances; we need a better process,” that’s a sign that production skills haven’t yet brought us to a fully mature commercial practice.

Progress toward an Engineering Discipline
The previous sections describe major shifts in technology and the research community’s focus. The question remains, are we any closer to an engineering discipline?

In “Prospects,” I identified several steps along software engineering’s path to becoming a true engineering discipline. They addressed the relationship between engineers and the substantive knowledge of the field. Here too, there’s been progress, but not all of a kind I anticipated.

Most notably, the democratization of the Internet has reshaped the role and nature of the engineering handbook. In “Prospects,” I argued for a better basis for routine practice of solving precedented problems, especially by improving access to knowledge about existing solutions. I imagined investment in better reference material, especially online analogs of classic engineering handbooks—centrally edited, definitive, and validated reference materials with a carefully designed structure and index to shape our understanding and inspire trust through their history and authorship. We now have substantially improved access to reference material, but in a form far from what I expected. Software developers keep all manner of documents online. These documents are indexed for search; a software developer with a question performs a search or queries a community of other developers. The reference handbook has become disintermediated! As with other crowd-sourced content, this trades away authority, accuracy, and organization in favor of extensiveness and currency, obliging each user to validate the information.

“Prospects” also identified professional spe-
cialization as a step toward maturity. This trend continues, although I now see three types of specialization rather than the two of “Prospects.” We see the predicted internal specialization in specific techniques such as communications, real-time embedded systems, graphics, and so on. We also see the predicted external specialization in application areas such as automotive systems and e-commerce. Unpredicted by “Prospects,” though, is an emerging opportunity to specialize in problem classes that occur in multiple domains and can be solved with different techniques—for example, adaptive-systems problems.

I close with a note on measuring progress. Researchers often cast “the software development problem” as a challenge to be “solved.” This is the wrong viewpoint. Our aspirations grow faster than our capabilities, so I don’t expect software development to “get solved.” Rather, I believe we’ll always be building systems at—or beyond—the hairy edge of our abilities, which will inevitably lead to unexpected difficulties. At the same time, we’ll find ways to create routine solutions to problems that were recently at our limits, as we’ve consistently done over time.

We should judge our progress not by whether we still struggle with our most ambitious projects but rather by the expansion of the “hairy edge” envelope and the growth of the classes of problems that we solve routinely. Engineering requires a genuine commitment to finding good solutions to clients’ problems. We should judge our progress toward an engineering discipline by judging the engineering sensibilities and techniques we use to solve both the hairy edge and the preceded problems—and to recognize in which class a given problem falls.

References