Editor: Robert L. Glass ■ Computing Trends ■ rglass@indiana.edu

Do We Know Enough to Teach Software Engineering?

Nicholas Zvegintzov

… in which I argue how software engineering education can become less vacuous

There's a nasty little secret about software engineering that books and journal articles don't speak of: most materials on software engineering do not actually teach the engineering of software. ("Engineering" here means the reliable, trustable, systematic building or modification of software to do useful work.)

In fact, do we know enough to teach software engineering at all?

In the July/August 2002 *IEEE Software*, two young writers dared to expose this secret. In "Software Engineering Is Not Enough," James Whittaker, a computer science professor at the Florida Institute of Technology, and Steven Atkin, an IBM engineer, defended practitioners who are often blamed for not learning enough about software engineering:

> *Much software engineering literature begins with the admonition that practitioners aren't doing enough—that the state of the practice is creating bad software. We do not dispute this fact. However, we believe that what software engineering literature offers as solutions are also not enough. Books on the subject favor the "light" side of the discipline: project management, software process improvement, schedule and cost estimation, and so forth. The real technology necessary to build software is often described abstractly, given as obvious, or ignored altogether. But software development is a fundamentally technical problem for which management solutions can be only partially effective.*

How justified is this criticism?

Whittaker and Atkin gave a deadpan but deadly critique of the amount of actual software in the two best-sellers of the current software engineering literature:

> *Finally, you decide that you simply read the wrong section of the software engineering book, so you try to find the sections that cover coding. A glance at the table of contents, however, shows few other places to look. For example,* Software Engineering: A Practitioner's Approach, *McGraw-Hill's best-selling software engineering text [by Roger Pressman], does not have a single program listing. Neither does it have a design that is translated into a program. Instead, the book is replete with project management, cost estimation, and design concepts.* Software Engineering: Theory and Practice, *Prentice Hall's best-seller [by Shari Pfleeger], does dedicate 22 pages to coding. However, this is only slightly more than four percent of the book's 543 pages.*

Continued on p. 110

0740-7459/03/$17.00 © 2003 IEEE

Continued from p. 112

The disciplines that Whittaker and Atkin brashly called "light"—project management, software process improvement, schedule and cost estimation, and so forth—do indeed fill out such books. But they are not in themselves the engineering of software—they are the mechanisms for controlling, managing, and packaging the engineering of software. According to Whittaker and Atkin, "software development gets so little attention in the software engineering literature because it is the hardest and least understood part of the software engineering life cycle." Seen from this perspective, software engineering is like the cartoon that shows two scientists in front of a vast blackboard covered with a long, complex formula. One of them is pointing to a small area in the middle, in which are the words "then a miracle occurs."

We know that this miracle occurs repeatedly. The world now depends on a profusion of software of amazing complexity that routinely does its job correctly—handles our finances, flies our airplanes, treats our diseases, and so on.

Practitioners have engineered this software and continue to engineer more of it. Did they learn to control, manage, and package this engineering from software engineering books and classes? Maybe, little by little. Did they learn how to engineer it from software engineering books and classes? Definitely not.

The necessary knowledge

To teach software engineering, we would have to conquer two problems:

- We would have to know how software is created and modified.
- We would have to write down such knowledge clearly and extensively enough to teach it.

So, what do we know about how software is created and modified?

Whittaker and Atkin offered a good summary of two bodies of knowledge practitioners must acquire to engineer software: "familiarizing themselves with the problems they are to solve (we'll call this *problem-domain expertise*) and studying the tools they will use to solve them (we'll call this *solution-domain expertise*)." (They add, "Software engineering doesn't help with [either] domain.")

In the problem domain, practitioners must understand finances, airplane aerodynamics, or human bodies (and what the software is intended to do with those things). This is real-world knowledge, not software knowledge.

In the solution domain, practitioners must understand the implementation technology—the language or languages in which the software will be expressed, the libraries and operating systems it will run on, and the development and testing platforms on which it will be composed. They must also understand all the real-world limitations (buffer lengths, timing considerations, and so on) of the finite implementation of these virtually ideal constructs.

Whittaker and Atkin provided sidebars with hilarious but hair-raising examples of the challenges of this knowledge. "The Making of a Maintenance Nightmare" exposed a single parameter (that was "published on a popular Web repository") that turned out to be a hive of at least 29 subparameters, many of them probably added ad hoc through time. "Plenty of Opportunities to Fail" cited Microsoft PowerPoint ("a large, complex application for making presentations and slide shows"):

> *When PowerPoint opens a file, 13 [Windows] kernel functions are called nearly 300 times; when it changes a font, two kernel functions are called a total of 10 times.*

Trying to describe problem-solution integration knowledge turns the best teachers into babblers.

But wait, there's more. Somehow (there's that miracle again) practitioners must integrate their knowledge of the problem domain's needs with the solution domain's resources, to implement the former in the latter. Call this *problem-solution integration expertise*.

Why textbooks don't teach

So how do software engineering textbooks teach these three domains? Not very well.

First, problem domain knowledge is outside the realm of software—it's real-world knowledge.

Second, much of the solution domain knowledge is found in programming-language manuals, system documentation, and library manuals. (Whittaker and Atkin: "The Win32 application programming interface contains over 20,000 calls. The printed reference manual for just the calls and their parameters is over four inches thick.") These are omitted by textbooks as too specific, too detailed, and maybe even too proprietary. Worse, much solution domain knowledge is picked up only in the field, in the programmer cubicle, as experience of what works and what doesn't or as borrowings from existing software, and therefore is not captured in textbooks.

Third, and worst of all, trying to describe problem-solution integration knowledge turns the best teachers into babblers of platitudes and tautologies: "You, um, learn the problem; you, um, learn the implementation; and then you, um, connect the two." Not very impressive verbiage to fill the core of a software engineering textbook.

The quest

But if we can't teach it, how can we learn it?

Here the news must be better, if only because we know that hundreds of thousands—maybe millions—of competent practitioners have learned it.

How do they actually do it? They use the internal resources of human intelligence—problem solving, insight, and a grasp of process. These resources are fueled by interest, hard work, and concentration; by know-how gathered from experience; and by reliance on existing software resources.

Observation of software practitioners does not show them emerging fully competent from software engineering education or training programs. In fact, programmers still come from diverse backgrounds. Observation of them at work shows a pattern of problem solving. Programmers' visible action is a discontinuous alternation of looking, mumbling, puzzling, discussing, and sometimes fragmentarily diagramming or scratching on paper, with occasional bursts of typing (and cutting and pasting). The typing is not programming; it only indicates achievement of a piece of programming. The programming is actually in the quest.

The quest is a probe through the problem space, continually forming subgoals—"What do we do next? What happens if we invoke this method? What does this parameter do? What happens to this data structure?" The practitioner is using his or her intelligence to find and then build linkages between the problem domain and the solution domain. As observers have noted, programming is an endless struggle with How? Why? What? Whether? and When? Typing takes place when enough questions are tentatively answered to permit the programmer to put a structure in place.

Little by little, empirical researchers are gathering more knowledge about how programmers work. Unfortunately for the teaching of software engineering, this is not yet the comforting imperative knowledge that methodologists aspire to ("Draw a box labeled A, draw another box labeled B, connect A to B with a double-headed arrow"). It is the frustrating descriptive knowledge of "Now she's looking at this. ... Now she tries that. ... Now she traces backward. ..."

This process, although unsatisfactory to the software engineering teacher, does get software engineered. And, better still, it builds better software engineers. Experience with the problem domain, a long and wide exposure to programming languages and programming platforms, and familiarity with already-built systems creates practitioners who build better software more quickly.

And the legacy of existing software itself provides a platform to build higher. Where once we struggled for days to put crude marks on a screen, now we can all make a radio button pulse on the screen by invoking a Windows method.

So, the answer to the question we started with—"Do we know enough to teach software engineering?"—is bad news; we don't. But the answer to a slightly different question—"Do we know enough to *learn* software engineering?"—is good news; we do.

A prescription for teachers

So what should the Pressmans and Pfleegers and the other not-yet best-selling teachers of software engineering be doing?

First, don't abandon the project management, software process improvement, schedule and cost estimation, and so forth, that currently fill software engineering books. Control and management will always be needed.

Second, accumulate and pay attention to software engineering's actual building materials—the algorithms, patterns, artifacts, and system manuals. Studies of the architecture of actual systems are important. The patterns literature shows a path because it links algorithms to the context and reason for using them. Instructions for testing, exercising, and debugging systems are practical and concrete. Techniques for expressing real-world requirements and for proving (or at least showing) that they correspond to system behavior are vital bridges. Four-inch-thick library listings are ammunition.

Third, follow the painfully acquired details of empirical studies as researchers look at how practitioners build, analyze, and modify software. Eventually this will pile up to prescriptive processes of how future practitioners should build, analyze, and modify software.

Fourth, teach by doing. Software engineering is not book learning. Teach with problems and projects where students (and teachers) labor to build and modify nontrivial, even real, systems in working groups.

Before we can teach we must learn. One day a unifier will gather all these materials together to make them concrete, and then we will know enough to teach software engineering.

Nicholas Zvegintzov is the president of Software Management Network (www.softwaremanagement.com), which supports software maintenance and supplies software service management expertise. Contact him at zvegint@attglobal.net.

Circulation: *IEEE Software* (ISSN 0740-7459) is published bimonthly by the IEEE Computer Society. IEEE headquarters: Three Park Ave., 17th Floor, New York, NY 10016-5997. IEEE Computer Society Publications Office: 10662 Los Vaqueros Cir., PO Box 3014, Los Alamitos, CA 90720-1314; (714) 821-8380; fax (714) 821-4010. IEEE Computer Society headquarters: 1730 Massachusetts Ave. NW, Washington, DC 20036-1903. Subscription rates: IEEE Computer Society members get the lowest rates and choice of media option—$43/34/56 US print/electronic/combination; go to http://computer.org/subscribe to order and for more information on other subscription prices. Back issues: $10 for members, $20 for nonmembers (plus shipping and handling). This magazine is available on microfiche.

Postmaster: Send undelivered copies and address changes to Circulation Dept., *IEEE Software*, PO Box 3014, Los Alamitos, CA 90720-1314. Periodicals Postage Paid at New York, NY, and at additional mailing offices. Canadian GST #125634188. Canada Post Publications Mail Product (Canadian Distribution) Sales Agreement Number 0487805. Printed in the USA.