Philippe Kruchten ■ University of British Columbia ■ kruchten@ieee.org

The Biological Half-Life of Software Engineering Ideas

Philippe Kruchten

I am pleased to inaugurate a new department by one of our most veteran editors. Philippe Kruchten is among the rare pioneers who have successfully straddled industry and academia. His contribution to software practice has been huge. He is now a professor of software engineering at the University of British Columbia in Vancouver, Canada. He spent 17 years at Rational Software, now part of IBM, where he led the development of the Rational Unified Process, a Web-based, generic software development process. He wrote three books on the RUP and created a model for representing software architecture based on multiple coordinated views, which led to an IEEE standard.

The insights brewed through his years of experience are sharp and penetrating. Thus, look forward to Philippe's and his guest authors' take on software engineering education, training, learning, certification, accreditation, and career advancement in these pages. From time to time, you'll also find in them invaluable information on the IEEE Computer Society's professional advancement initiatives. Oh, before I forget … expect Philippe to touch sensitive chords and provoke thoughtful discussion on hairy issues pertaining to the development of our profession. Enjoy the first edition! —Hakan Erdogmus, editor in chief

A product's biological half-life is the time it takes the body to eliminate one half of the product taken in by natural biological means. For example, caffeine's half-life is roughly three and a half hours. Of all the molecules of coffee in the cup I just finished, my body will have eliminated—or broken down into simpler compounds—half in three hours, three quarters in six hours, and so on.

Using the same general idea, I've often wondered about the half-life of important software engineering concepts, tools, methods, and even companies. If you were to compose a list of 100 important concepts in year T_0, how many would still be important in year $T_0 + N$?

The five-year hypothesis

My conjecture is that the half-life of software engineering ideas is roughly five years. Five years from now, 50 percent of the key ideas, concepts, and so on in this copy of *IEEE Software* will have been forgotten or seriously marginalized—not really worth teaching an undergraduate software engineering student, for example. No, I haven't rigorously tested this hypothesis, but just for fun, I took from my shelf a few issues of *IEEE Software* from 1988 (this shows my age, I know). What do we have here? Lots of articles about programming languages: Fortran (okay, it's still around in some circles, but not taught much), Eiffel (a small niche of fans), Ada (very marginal; gee, I loved that language, so here I'll drop an emotional tear), TurboPascal (yep, used that), someone who wants to integrate Loops with Prolog, and a visual front end for Prolog, touted as the new great way forward.

On the operating-systems front, OS/2 is mentioned most (gone now). And we have companies buying full-page ads: Stepstone (gone), Softool (gone), and Interactive Software (morphed). On systems, the hypercube computer is the state of the art, and rapid prototyping is the new "in" process (which, in some ways, survives in early iterations

 0740-7459/08/$25.00 © 2008 IEEE

of agile development). Out of 50 items I checked, maybe three are still important today, or at least recognizable. That's indeed a half-life of five years. This made me wonder: of 200 things I learned about software in school, only one or two would still be key ideas today! What could those be? Modularity? Synchronization between processes? The Dijkstra/Parnas/Hoare stuff?

Keeping up-to-date

So, if concepts, ideas, tools, or techniques in our field have a half-life of five years, we need to constantly replenish our brains' content. We can't stop learning new things, or we'll get empty pretty rapidly, and we'll be totally useless, obsolete, hit by The Peter Principle. We must constantly learn new tricks. We snooze, and poof, we're off the wagon. The next time we look for a job, we won't even recognize what the ads are talking about.

We also have an ethical duty to keep up to speed with advances in our field. The *IEEE/ACM Software Engineering Code of Ethics* (you've read it and have a copy handy on your hard drive, right?) states that

> *Software engineers shall participate in lifelong learning regarding the practice of their profession. … They shall continually endeavor to further their knowledge of developments in the analysis, specification, design, development, maintenance and testing of software …, together with the management of the development process, … to improve their knowledge of relevant standards and the law governing the software and related documents.*

Professional organizations worldwide have started to take a stand about this, seeing that their members take a rather relaxed and lazy view of the topic, and they now mandate Continuous Professional Development (CPD). If you happen to be an IEEE Certified Software Development Professional (CSDP; I'm #99), you know that every five years (five years, huh?) you must demonstrate that you've kept up-to-date, in some way or other: taking classes, reading books, going to seminars and conferences—I get a few points for researching and writing this article! Engineers Canada and its constituent bodies have taken a similar approach; I must now tick a box on my yearly license renewal as a professional engineer, and in a few years, they might even audit my CPD record.

During parts of my career, I was heavily involved in hiring new people for large software teams. Some of my questions often puzzled candidates: What technical book have you read in the past six months? What technical publication do you subscribe to or read regularly? Tell me about one new thing you've learned lately, a new idea that you haven't yet tried, but are eager to? I've often been disappointed by the results. I didn't give much of a chance to someone who hadn't read a single book, or sometimes a single technical publication, in six months.

Now that I'm retired from industry and teaching software engineering, I often get into debates with students about the training-versus-education issue. One came to see me with a long list of languages, tools, and techniques (a long acronym soup) picked from all the ads he could find for software positions, and said, "We are not learning the right things here; we should be learning C#, RUP, Ajax, Perl, Python, DB2, HTML, Oracle 9, SAP R5," and so on, and my reply was invariably this: How about you learn how to learn? Wouldn't that be more useful than just learning a large collection of techniques that are going to vanish soon (half-life: five years)? How about getting into your system some more complex molecules that don't dissolve too rapidly, that have half-lives of 20 years?

So, we need to learn continuously. But how do we go about doing this? You seem to be reading *IEEE Software*—is this sufficient? What if you don't live in or close to a school? Or a library? There's the Net, of course: Google, Slashdot, the blogosphere, RSS feeds, and wikis. But how do we sort out the anecdotal, the crazy opinions, the sales pitches, and the rants from solid and validated information? Where to go? Or not to go? How to avoid being swamped?

Professional development

This is the first installment in a new *IEEE Software* column on professional development. In small doses over the next few months, we'll be covering in more detail some of these issues: body of knowledge, knowledge transfer, education and curriculum, continuing education, certification, professionalism, associations, career advancement, and the relationships between all these things. I don't intend to pontificate alone on all these topics, and I'll find knowledgeable helpers in my personal network to write some of this—in fact, consider this a call for participation.

If some of these issues are dear to your heart, if *you* want to share them with us, I'd be happy to hand over the microphone (or the keyboard) for 1,500 words. But contact me first, and let's discuss your idea.

Philippe Kruchten is a professor of software engineering at the University of British Columbia in Vancouver. Contact him at kruchten@ieee.org.