Editor in Chief: Hakan Erdogmus ■ Kalemun Research ■ hakan.erdogmus@computer.org

Déjà Vu: The Life of Software Engineering Ideas

Hakan Erdogmus

The story of software engineering since the label came into use is thus a story of compromise among generality and specificity, heuristics and formalism, procedures and data, sequence and cycle. The practical response was combination and accommodation—covering all bases or splitting the difference, synthesizing complementary approaches or accommodating inescapable trade-offs. Pragmatists argued for mixed strategies of testing and proving, the use of tailored reliability models and development environments, the use of a full set of metrics, and the synthesis of life-cycle models. But while seizing the middle ground appeared to be a practical way to cope with difficulties, it seemed unlikely to produce a revolution. If software technologists are nowadays devoting more effort to engaging in a pragmatic fashion with the complexity of their problems, it is to their credit. That is symptomatic of maturity and of real engineering. —Stuart Shapiro, "Splitting the Difference: The Historical Necessity of Synthesis in Software Engineering," *IEEE Annals of the History of Computing*, vol. 19, no. 1, 1997

I was scanning through notable scientific discoveries of 2008 on the net (www.wired.com/wiredscience/2009/01/top-10-scientif). They included discovery of ice on Mars, the Cray XT5 supercomputer that broke the petaflop barrier, nanotube paper that is lighter and stronger than steel, and a catalyst that efficiently turns water into fuel. I wasn't surprised that the list didn't include anything that I would associate with software engineering. But it still struck me that even if a recent hot trend in software engineering had made it to the list, it wouldn't have been possible to communicate it to the lay software person without a lengthy, long-winded explanation. The reader would lose interest before the first sentence was over.

Each of the discoveries I just mentioned qualify as "rocket science:" they're pretty deep. Software engineering innovations can also be hugely complex, but often you can't package them neatly to be as self-evident as "the catalyst that efficiently turns water into fuel." Modern software engineering ideas tend to be deep in a different way: they're subtle and intangible, they're concept-rich, they intermingle, and they go through cycles in a fast and ever-changing environment.

The Nature of the Field

The most liberal definition of software engineering that I can think of without too much blurring of the boundaries with related disciplines would go something like this: software engineering is building good and sustainable software using systematic, sustainable, and economic means. By extension, a broad definition of software engineering innovation would be the discovery and accep-

Mission Statement: To be the best source of reliable, useful, peer-reviewed information for leading software practitioners—the developers and managers who want to keep up with rapid technology change.

New Column: Impact

In this issue, Michiel van Genuchten and Les Hatton inaugurate a new department on software's impact. Michiel, Les, and their guest contributors embark on the ambitious task of demonstrating the ubiquitous presence of software in diverse industries. Their arsenal: experience and data from the trenches. Their message to those who think software is just another invisible, annoying overhead: software is under every stone, software grows beyond your imagination, software may be unsustainably expensive, and yes, software is critical to your business. Enjoy!

Erratum

In the November/December 2009 edition of this column ("A Process That Is Not"), the sentence at the end of the first paragraph "Yet Tiki works, despite the software and antiprocess used in developing it" on page 4 should have read "Yet Tiki works, despite the antiprocess used in developing it." I apologize for the mistake.

Kudos

Columnist Grady Booch was elected an IEEE Fellow for his contributions to software engineering and the development of the Unified Modeling Language. Congratulations on this important recognition.

tance of those means. Here I deliberately avoid the usual, more righteous definition that emphasizes the use of scientific and mathematically based means. The way we build software involves technology in many instances, but that alone isn't sufficient. Regardless of whose definition, how technology is applied and why it works are deemed critical (sometimes even more critical) in the software engineering world. Thus such means also centrally involve tenets, methods, practices, and understanding: in effect, the "how" and "why" part of building software. All of this smells a lot like process, and perhaps software engineering is by definition a "processy" field, softer than we'd like it to be. If that's true, we shouldn't expect frequent step-function improvements. The inherent process focus may also be why, in his article "Software Engineering: An Idea Whose Time Has Come and Gone?" (*IEEE Software*, July/August 2009), Tom DeMarco appears to equate software engineering with the process side of software engineering. In that article, DeMarco recants some of his own legacy advice about the importance of measurement and control in software projects, but in doing so, he also implicitly overgeneralizes his impressions about the process aspect to an entire field.

We should finally resolve that the discovery of new software engineering ideas is, by now, naturally incremental and evolutionary. This insight is not novel at all. Fred Brooks famously told us so over 20 years ago ("No Silver Bullet—Essence and Accidents of Software Engineering," *Computer*, April 1987) when he identified complexity, conformity (arbitrariness of that complexity), changeability, and invisibility as software's four essential difficulties. Stuart Shapiro later referred to the incremental nature of software engineering discoveries in his excellent historical account "Splitting the Difference." Lawrence Peters and Leonard Tripp described designing software as a "wicked problem," a changeling that doesn't lend itself to a clean, stable solution ("Is Software Design Wicked?" *Datamation*, May 1976, p. 127). The key ideas—among them, abstraction, modularity and information hiding, reuse, better communication, and attention to human aspects—for dealing with essential difficulties have been around quite a while.

The Intertwined Lives of Ideas

Suppose we've dampened our expectations of step-function improvements and moved

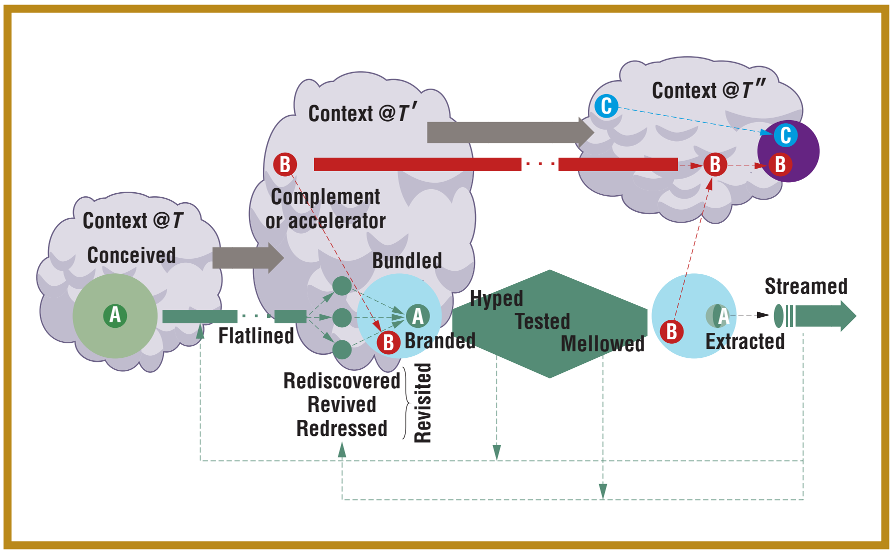

Figure 1. The life cycle of modern software engineering ideas.

on. What about ideas that have merit but lead to incremental improvements? Are they readily recognized, accepted, and ultimately adopted? The short answer is, no. If payoffs aren't obvious and certain, then acceptance, naturally, is also elusive. Fortunately, some factors that contribute to this elusiveness are partially within the reach of our control. We can start looking for them in the iterative and interleaved cycle in which ideas are repeatedly visited, bundled, branded, spun off, and moved across each other's pipelines. Once the stages and states of that life cycle are better understood, underlying acceptance and adoption levers can be identified. Alas, there's the obvious caveat: if modern software engineering is inherently incremental, we shouldn't have too high expectations of those levers either.

Figure 1 illustrates the life cycle of a typical software engineering idea in the post "Silver Bullet" era.

Dormancy to Reincarnation

An original idea is *conceived* in an initial context at time *T*. The context is the general environment in which the idea is positioned. It includes the problems being addressed, related ideas, counter-ideas, accelerators and decelerators, target audience (receptors and users), surrounding technologies (both established and emerging), and centrally, the relevant technical, social, cultural, and economic conditions.

The idea often is not disseminated by itself but is part of a larger package, or *bundle*. Think of the bundle as the immediate context that the idea's advocates deem necessary to make the idea operationally viable and allow it to be applied in practice. The bundle is the collection of vital complementary and synergistic concepts, approaches, practices, technologies, and norms—in short, all the things that the proponents think will make the idea work.

After its initial exposure, the idea may stay dormant for a while: in this state, it's effectively *flatlined*. It doesn't get much attention: it's inconspicuously used by a small number of people, usually by the inventors. Flatlining might happen for various reasons. The idea might still be ahead of its time: the general mindset necessary for the idea to take off might be lacking. It might not have a name and sufficient surrounding vocabulary, resulting in poor awareness and understanding. The idea might be perceived as straightforward and thus unimportant. Or a key supporting technology might be missing.

If the idea has merit, it doesn't stay dormant forever. The context eventually evolves and the idea ripens with it. Soon after, the idea is *revisited*, sometimes by the original founders but more often by others. Reincarnation occurs in one of three ways: the idea may be revived in its original form, independently rediscovered close to its original form, or deliberately redressed for the new context. The new context at time T' likely includes a complement (a supporting concept that in the re-inventers' eyes makes the idea whole) or an accelerator (a synergistic concept that facilitates the idea's application). The original bundle is unraveled, and the idea is rebundled with its new complements and accelerators in the new context. The resulting whole is *branded*, using representative vocabulary that is aligned with the new context and giving the idea a fresh spin. At this point, books are written about and around the idea.

Ascent and Descent

Bundling and branding injects the idea with new life, triggering its growth-and-decline phase. This phase roughly corresponds to Geoffrey A. Moore's technology adoption life cycle (*Crossing the Chasm: Marketing and Selling High-Tech Products to Mainstream Customers*, HarperCollins, 1999). The main difference is that the end result of this cycle might be weaker than mainstream adoption. If incremental improvements are the norm, we should expect mainstream adoption to be rare. Rather, the outcome may more modestly be general acceptance by a target audience coupled with adoption in a specific context. Reversion to a previous state is possible during the growth-and-decline phase.

Once the idea is branded inside a new bundle, its real ascension begins. Its advocates *hype* the idea. It enjoys rapid spread and rise in popularity as it's picked up by the target audience. More and more people start paying attention, including vendors looking for concepts to leverage in new products and services. The idea becomes available for sandboxing in various forms, and eager early adopters start using, and effectively *testing*, it. As its usage increases, more scalable forms become available, and efforts to make it work intensify. Sometimes it's applied outside the intended context and in unexpected ways. Users discover and publicize the idea's limitations and new contexts, and modify it along the way. During the testing stage, the idea goes through a natural selection process.

Sobering to Stability

As its limitations become widely known, the idea *mellows* by stripping off its nonessential and dysfunctional aspects. The hype gradually dies down but the essence, the modified core that deserves genuine merit, remains. That core is subsequently *extracted* from the bundle. Ultimately, the part that catches on is the extraction: typically, the target audience accepts the idea's distilled form and adopts it in practice, independent of its former bundles. I refer to this post-extraction state as *streamed*. In the streamed state, the idea enjoys a stable existence and need not be actively promoted.

The weaker form "streamed" instead of "mainstreamed" communicates the subtlety that "main" is relative. Acceptance and adoption are situational in most cases: one size almost never fits all.

I have clumped acceptance and adoption together as if they're the same thing. They need not be. An idea is accepted when it's widely recognized as a viable solution to a problem in a given context. An accepted idea has a name and an associated vocabulary, though the name may eventually fade into the background. If complex enough, an accepted idea supports a stable community of experts. Adoption happens when an idea is established enough to the point that it's frequently applied in that context to solve the underlying problem. Adoption is a natural progression of acceptance. If adoption is preempted, a streamed idea ultimately suffers reversion.

Reversion and Threading

The feedback loops in Figure 1 convey the possibility of reversion. Reversion may happen at any point during the growth-and-decline phase or after the idea has been streamed. It results from the introduction of an alternative idea that disrupts the cycle, a change in contexts that invalidates the preconditions for acceptance, or unfavorable anecdotes from the trenches. These triggers cause the idea to regress rapidly and ultimately be flatlined. A regressed idea can be revisited in a new context to regain consciousness. While reversion makes the maturation process iterative, the effect is not necessarily a productive one.

During the lifespan of one idea, another related idea starts a parallel cycle of its own. Figure 1 depicts an instance of this phenomenon for two ideas, A and B. The complement or accelerator B, which was pulled into A's bundle from A's context at time T', becomes more prominent during A's growth-and-decline phase. It flatlines outside A's context from T' to T''. By T'', B has gained enough prominence, and in the new context of its own, it becomes ready to be bundled and branded with a third idea C. The result of such interaction is a complex network of parallel pipelines, with ideas moving back and forth across different tracks. I refer to this property of the idea life cycle as *threaded*.

In a Nutshell

The progression of software engineering ideas appears to follow an incremental, reversive, and threaded process. I captured these characteristics in a life-cycle model. The model itself is an attempt to provide a singular perspective on a complex phenomenon. Therefore, we must take it at face value. Different ideas go through the maturation process in different ways. Some may crawl through it, lingering in certain states. Others may zip through it, bypassing entire stages. When I described the model in a recent talk, colleague Tim Lethbridge rightly mentioned modern program control structures (over the demoded go-to statement) as an example of the latter kind, an obviously good and applicable idea with swift and permanent acceptance.

My goal was to implant the thought of pursuing progress within the distinct possibility that incrementality, reversion, and threading are natural in software engineering innovation. Taking advantage of whatever levers we can find underneath these characteristics may be the best we can do for streaming a good idea without surrendering to reversion. My space is up, so those levers, and several examples, will be the topic of future columns. Stay tuned.

Acknowledgments

Participants in the November meeting of the Consortium for Software Engineering Research provided me with ample feedback on this column's topic. Luiz Capretz of the University of Western Ontario pointed me to Stuart Shapiro's well researched and highly relevant article.

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; +1 714 821 8380; fax +1 714 821 4010. IEEE Computer Society headquarters: 2001 L St., Ste. 700, Washington, DC 20036. Subscription rates: IEEE Computer Society members get the lowest rate of US$52 per year, which includes printed issues plus online access to all issues published since 1988. Go to www.computer.org/subscribe to order and for more information on other subscription prices. Back issues: $20 for members, $163 for nonmembers (plus shipping and handling).

Postmaster: Send undelivered copies and address changes to *IEEE Software*, Membership Processing Dept., IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08854-4141. Periodicals Postage Paid at New York, NY, and at additional mailing offices. Canadian GST #125634188. Canada Post Publications Mail Agreement Number 40013885. Return undeliverable Canadian addresses to PO Box 122, Niagara Falls, ON L2E 6S8, Canada. Printed in the USA.