Breaking Out of the Software Engineering Mind-Mold

Terry Bollinger, MITRE

Do as you’re told, and all will be well!” Or will it? Read on if you dare to question the status quo.

Discipline and freedom

I like to read anything that makes me think. As an example—I recently read an article proposing that software engineering is an attempt to merge two opposing themes. Paraphrasing a bit, I would call the first theme a discipline of completeness; the second is freedom of creation.

The discipline of completeness is a willingness and ability to search for and deal not just with every conceivable mode of failure but with as many inconceivable modes as you can find through exploration and testing. Nonprogrammers often have less complementary ways of describing this notion, such as “Wow, you sure are a nitpicker,” or “How can you stand to keep looking at that program?” or even “Do you have an obsessive–compulsive disorder or something?” And indeed, this discipline of completeness would be excessive in some contexts. But software happens not to be one of those contexts. When computers can amplify even a tiny mistake into, say, an outage of the telephone systems on the east or west coast, the positive benefits of being a devoted nitpicker increase rather dramatically. The remarkable ability of modern networked computers to amplify tiny errors into global catastrophes makes the discipline of completeness a vital and fundamental theme of any realistic concept of software engineering.

The opposing theme is freedom of creation, as in: If you are not a highly creative person, why in the world are you writing software? After all, all the mindless plug-and-chug tasks in
software development died with the creation of the first stored-program computers in the 1940s. (Nevertheless, by most accounts, the corpse of mindlessly repetitive programming continues to thrash about with gusto in development organizations around the globe.)

When you build machines out of pure information—that is, when you write software—any good reason for solving the same problem twice pretty much disappears, since that solution can in principle be applied wherever and whenever it is needed in the future. There are, unfortunately, many persuasive but bad reasons to keep people reinventing the wheels of software infrastructure. Some of these are entirely bogus, such as the need to generate impressive process improvement statistics, job security, fear of the unknown, inadequate and nonexistent sharing of results, Not Invented Here attitudes, or wrong people in the wrong jobs. Other reasons make sense in the short term but reflect larger problems that desperately need to be fixed. These include unacceptably low quality, incomprehensible documentation, and software architectures that encourage repetitive or replicated maintenance activities. Like false tracks branching off a mountaintop trail, however, rationales for applying creativity to replication are ultimately research and market dead-ends.

Breaking out of the box

I hope that more than a few of you are thinking, “I’m with you on the first point, but how can you possibly reconcile the discipline-of-completeness theme with all that radical and scary creativity stuff?”

I’m glad you asked. Your question leads right into this IEEE Software focus section, in which bold and insightful authors from around the globe dare to take on the difficult question of why software engineering all too often doesn’t work as well as promised. If, for example, you would like to read what inspired my earlier musings, take a look at the powerful and informative article “Improving Software Process Improvement” by Reidar Conradi of Norway and Alfonso Fuggetta of Italy. This international team dares to question the assumption that simply changing a process automatically means you are improving it. They show how embracing the often-overlooked maintenance phase as an integral part of the problem can lead to better designs and improved long-term cost control.

Another personal favorite of mine is “Software Engineering Is Not Enough” by James A. Whittaker of Florida and Steven Atkin of Texas. These authors dare to question the very fabric of software engineering; they challenge the casual dismissal of programming as a “minor” component of the overall software process. Anyone who thinks that choosing programmers is less important than choosing a software process should read this article carefully.

Along the lines of providing better understanding and insight into the complexities of software processes, we have “Software Measurement: Uncertainty and Causal Modeling” by Norman Fenton, Paul Krause, and Martin Neil. These London authors propose using Bayesian networks to model and understand the risks associated with software development.


I hope you enjoy reading this thought-provoking collection of articles as much as I did. I expect that one or more of them will inspire you to look at software engineering in a new light. 

---

**About the Author**

**Terry Bollinger** is a principal information systems engineer at The Mite Corporation, where he gets to work in topics that range from software interoperability and distributed architectures to quantum computing and nanotechnologies. He is IEEE Software’s associate editor in chief for software construction methods, primary author of the software construction section of the draft Software Engineering Body of Knowledge (SWEBOK) standard, and a winner of the IEEE Millennium award. He has a BS and an MS in computer science from the University of Missouri, Rolla. Contact him at terry@mite.org.