Software Is Driving Software Engineering?

George Hurlburt and Jeffrey Voas

SOFTWARE ENGINEERING is quite well defined. In 2014, the IEEE Computer Society released the third edition of its comprehensive Guide to the Software Engineering Body of Knowledge (SWEBOK Guide).1 Figure 1 shows part of the SWEBOK Guide’s conceptual layout. The boxes show major topics, with subtopics listed in the descending structures. Each subtopic is further broken down and supported by even deeper levels, leading to the textual treatment of everything.

Despite the SWEBOK Guide’s thoroughness and apparent currency, it faces one fundamental challenge. Software continues to morph and expand in influence with increasing rapidity.

Why must the SWEBOK Guide face continual change? It turns out we’re living in a physical world that’s moving at the speed of software. This means that software’s trajectory will drive software engineering, not vice versa.

A Brief History of Software

Consider the early software achievements, in which linear mathematics reigned supreme. Linear ballistic trajectory calculation was considered a triumphant achievement in the late ’40s. The US space program brought ever more dynamic mathematical navigation problems to the forefront, literally taking us to the moon. Relational database management systems began to overcome expensive storage constraints and brought transaction processing to businesses, thus fueling functional programming by the ’60s. PC-compatible operating systems brought computational power to individuals, incidentally vastly expanding the pool of potential programmers in the ’80s.

Lately, the Internet has pioneered the notion of a global network in which everything can be connected. Now, the Internet of Anything is rapidly extending this notion well beyond human networks.2 The huge mobile-device market is further reinforcing and hastening this network phenomenon. By 1989, the mobile phone packed more computational power than an Apollo mission computer.

Software has evolved similarly. Once considered a tool for rapidly and efficiently solving complicated mathematical problems, software has become a logical means to relate diverse ideas across vast networks. In so doing, software has migrated from mathematically precise expression to an environment in which meaning and data provenance often matter. It now supports expression of human abstractions understandable only in increasingly fuzzy functional contexts. Software can no longer be decoupled from the processes or functions it supports. As programming languages, such as Haskell, become more abstract, the question of precise meaning becomes increasingly urgent. Ontology is already becoming a prerequisite to disambiguation of semantic variation in which the relationships between plentiful software nodes are overwhelmingly many-to-many.3
Software has transited from standalone programs performing singular functions to deeply embedded control mechanisms in vast system-of-systems environments. The epitome of such an environment is the smart grid, in which many key variables, typically outside the system, are in constant flux—sometimes somewhat rhythmic and sometimes totally asymmetric.

Because software routines are deeply embedded in systems, they too become highly interdependent. Today, any software interaction suggests that there are multiple paths, all influenced by sensor input from the external environment, to achieve a given end. Consider an autonomous automobile informed largely by a constantly learning Bayesian network. The optimal solution for a given subsystem at one microsecond might differ significantly from subsequent solutions in succeeding microseconds.

So, cause-and-effect relationships relate to paths through multiple software modules as influenced by sensor input and feedback, not by any single program’s direct, discernable action. This argues against strict determinism, refutes reductionism as a valid software-testing approach, and drives any solution to nonlinear proportions. Indeed, software has moved standalone routines to adapt along with complex systems; in so doing, these routines have become complex adaptive entities in their own right. Embedded software’s nonlinearity further refutes the notion that we can engineer software, much less test it, in any classically linear fashion.

As systems of systems become further embedded in networks of networks, the potential for self-organizing behavior increases substantially. Consider a network of autonomous vehicles on grid-enabled highways. The realm of nonlinear decision points and potential paths will grow to mammoth proportions as the Internet of Anything advances.

**What Motivates Software Engineers?**

For future software to be managed effectively, it would appear that dynamic software interdependence reigns supreme. But does this mesh with the nature of software developers, who live in the moment?

**Monetary Gain**

Some people would assert that money motivates. Software engineers are generally well compensated. According to the US Bureau....
of Labor Statistics, in 2014, an application developer earned a median US$99,510, and a systems developer earned up to $154,800. Moreover, the bureau projected that employment of software developers would grow 22 percent from 2012 to 2022, clearly making software engineering an up-and-coming trade.

Fame
Others might assert that software engineers are in the game for notoriety. However, most software engineers tend toward the sensing, thinking, and judging side of the Myers-Briggs Type Indicator (MBTI), suggesting a tendency toward solitary behavior. Of the 94 individuals inducted into the CompTIA IT Hall of Fame from 1998 through 2014, only 19 percent had an appreciable programming background. Of them, only Paul Allen, Bill Gates, Mitch Kapor, Steve Wozniak, Vint Cerf, Grace Hopper, and Tim Berners-Lee had significant global name recognition. Fame appears difficult for individual programmers to achieve.

Maker Mentality
Software engineers are often motivated by the magic of making amazing things happen that improve the world through creative software. Software can reach far, while the creator, perhaps still in pajamas, never leaves the comforts of home.

Artistic Appeal
Because software ultimately depends on language, proper usage and syntax become important. Although appropriate language usage indeed fuels good discipline, the ability to express the same concept in myriad ways remains the choice of any language-savvy programmer. This refutes the scientific approach to software development because the ability exists to relay concepts either awkwardly or with the utmost eloquence. This choice tips the scale away from rigorous science and toward artful expressionism as a motivator.

Design Thinking
Advocates of STEAM (science, technology, engineering, arts, and mathematics) emphasize the artistic creativity released through masterful software design. Pointing to the need for innovative “design thinking,” they extoll the art of great software much like critics celebrate great novelists. Unlike the more literal view of software engineering, this view takes software development to a pinnacle of literate achievement, in which self-expression is the reward.

Rigor
In keeping with the prevalent MBTI profile, perhaps the mathematical rigor and discipline of the trade are motivators. However, the hypothesis that software engineering brings engineering precision to software development remains unproven. Tactics to enforce software development as an engineering phenomenon, including formalisms, strict languages, licensure, design patterns, or reusable components, have all fallen well short of their intended productivity gains. Nonetheless, mission-critical software requires a level of precision that clearly excludes purely artistic motivation.

Automated Software
Finally, electricity might be what makes seemingly efficient software engineering magic. Here, the “software engineers” are robotic systems that crank out code according to abstract human rules and parameters. Such large-scale code generators often produce predictable code in useful formats.

Blurring the Lines
The motivation for good software appears to lie somewhere between an art and a science, having attributes of both but belonging to neither. The lines blur further as software becomes more interdependent, diminishing the role of both the artist and meticulous software engineer.

What Really Drives Software Engineering?
On a trip to Boulder, Colorado, perhaps the hotbed of this decade’s entrepreneurial spirit, one of us visited the now-famous coffee shop, Atlas Purveyors (now permanently closed), the home of many new ventures. The programmer working intently on a laptop in the shop’s deep recesses was clearly off limits. This superhero was on task and untouchable. Subsequent meet-ups in Boulder revealed...
that this entrepreneur had good business ideas. Nonetheless, the ability to name a respected coding team was the real differentiator between just another good business idea and a truly supportable venture. The ratio of CompTIA IT Hall of Fame selectees who were clearly programming gurus to others reinforces the observed relationships.

CompTIA also selected individuals such as Steve Jobs, Steve Ballmer, and numerous other CEOs as recognized IT leaders. With diverse backgrounds typically not directly related to coding skills, these selectees made substantial contributions. The strategic thrust and determination of the corporate leaders, backed by software developers, were what signified enduring contributions to all aspects of the IT industry. Their vision and drive for business perfection, including artistic design features and unparalleled innovation, made the difference in outcome.

Perhaps software engineers are really highly skilled and valued tradespeople, whose role is shaped and focused by true leaders with a broad visionary reach. Recall the cathedrals of old with their flying buttresses and phenomenal masonry. Perhaps the new generation of stonemasons is the growing cadre of programmers whose cathedrals soar because their patrons, the visionary captains of industry, can conceive of the grand structure and drive its creation despite the odds. How else did Steve Jobs, among other feats, employ computer technology, including software, to transform the music industry? It will be a broad vision that will see us through.

**References**


**George Hurlburt** is the chief scientist at STEMCorp. Contact him at gfhurlburt@gmail.com.

**Jeffrey Voas** is a computer scientist at the National Institute of Standards and Technology. Contact him at jeffrey.m.voas@gmail.com.